Table of Contents

List of Figures

1. Organization of the Statice Documentation

Statice is an object-oriented database system for the Genera programming environment. Statice provides client programs with *persistent*, *shared* storage of information. Persistent information stored in Statice exists outside and beyond the boundaries of the Lisp world that created it, and is protected against failure. Shared information is shared by distinct Lisp worlds on different workstations, for writing as well as reading.

The Statice documentation is divided into several categories. The installation instructions are delivered along with the cover letter.

We start with a tutorial that covers the basic concepts in the context of example programs:

"Tutorial Introduction to Statice"

Statice is a powerful programming tool that includes many advanced techniques, most of which can be learned separately from the others. Any given application program will not need all of these techniques, so you can select from the next chapter those that sound most useful to your application:

"Advanced Techniques for Statice Applications"

Since performance is a key aspect of a database, we present a separate chapter discussing how to maximize the performance of your application:

"Statice Performance Issues"

The next chapter describes the administration of Statice:

"Operations and Maintenance of Statice Databases"

The next category is the reference documentation. We start by documenting the Statice commands. We then cover Statice functions, special forms, and macros. Each Statice operator is briefly mentioned in the Summary, and then documented fully in the Dictionary.

"Dictionary of Statice Commands"
"Summary of Statice Operators"
"Dictionary of Statice Operators"

The documentation presents several example programs. You can find the Lisp source of these programs in the directory SYS:STATICE;EXAMPLES;.

2. Tutorial Introduction to Statice

To show you how to make use of Statice in your programs, we present this tutorial of example programs. The tutorial is designed to be read sequentially. Within the tutorial you will see cross-references to more detailed documentation, but we recommend that you go through the tutorial in sequence, and postpone the cross-references until later. However, feel free to read it in whatever style suits you best.

2.1. Quick Overview of Statice: the Bank Example

2.1.1. Basic Concepts of Statice

We start with a simple example that demonstrates the basic concepts and facilities of Statice. The explanations are sketchy, designed to give you an overall idea of what Statice is, while leaving the details till later. The complete source listing of the example is given in this section and in the file SYS:STATICE;EXAMPLES;BANK-EXAMPLE.LISP.

How to Define a Statice Schema

In this example we define a bank database. A bank database is made up of accounts, and each account has a name and a balance. The following forms define the *schema*:

```
(define-schema bank (account))

(define-entity-type account ()
  ((name string :inverse account-named :unique t)
   (balance integer)))
```

The **statice:define-schema** form defines a schema named **bank**, and says that it has one *entity type*, named **account**. The **statice:define-entity-type** form defines the entity type named **account**, and says that it has two *attributes*, named **name** and **balance**. The **name** attribute is declared to be *unique*, which means that no two accounts can have the same name.

statice:define-entity-type is analogous to **defstruct** or **defflavor**, in that it defines a new type. The attributes of an entity are analogous to slots or instance variables. The main difference between entity types and types defined by **defstruct** and **defflavor** is that entity types reside not in a Lisp world, but in a Statice database. Another difference is that each attribute has a *type*. In this case, account names are strings, and account balances are integers.

The **statice:define-entity-type** form automatically defines *accessor functions* (for accessing the value of an entity's attributes) and an *entity constructor function* (for creating new entities of this entity type). We show examples of using these functions later on in this section. See the section "Defining a Statice Schema", page 7.

How to Make a Statice Database

We need to state where the bank database will be stored. Every Statice database is stored in a file in a *Statice File System*. The Statice File System might be on your own host or on another host on the network.

The following form defines a variable to hold a pathname, which we will use to indicate where the database is stored. This pathname is different from others because **beet** is not a host name, but the name of a Statice File System.

```
(defvar *bank-pathname* #p"beet:>finance>bank")
```

Below, we define **make-bank-database**. This function creates a new database in the place specified by the pathname, and initializes it to be a bank database with no accounts.

```
(defun make-bank-database ()
  (make-database *bank-pathname* 'bank))
```

If you are eager to plunge in and start using Statice, and want instructions on how to set up a Statice File System: See the section "Using Statice for the First Time", page 17.

How to Make New Statice Entities

When you define an entity type, **statice:define-entity-type** automatically defines an *entity constructor function*. For example, **make-account** is the entity constructor function for making new account entities. Below we define **make-new-account**, which makes a new account in the bank database by calling **make-account**, the entity constructor.

```
(defun make-new-account (new-name new-balance)
  (with-database (db *bank-pathname*)
    (with-transaction ()
      (make-account :name new-name :balance new-balance))))
```

See the section "Making New Statice Entities", page 12.

How to Access a Statice Database

Notice that the definitions of **make-new-account** (above) and the definition of **deposit-to-account** (below) use **statice:with-database** and **statice:with-transaction** when accessing the database.

The **statice:with-database** form is analogous to **with-open-file**. It opens the database (if necessary) and makes this database the *current database* during the execution of its body.

Every operation that examines or modifies a database must be done within the dynamic extent of a **statice:with-transaction** form. **statice:with-transaction** delimits a single *transaction* on the database. A transaction is a group of operations on a database, with the following properties:

- Each transaction accesses shared data without interfering with other transactions.

- If a transaction terminates normally, all of its effects are made persistent; otherwise it has no effect at all.

See the section "Accessing a Statice Database", page 9.

How to Access Information in a Database

To read information about an entity in the database, we use the *reader functions* that were defined automatically by **statice:define-entity-type** form. There is a reader for each attribute of an entity. In the bank example, the readers are called **account-name** and **account-balance**.

To write information about an entity in the database, we use the *writer functions* that are defined automatically. To call a writer function, use **setf** with the reader function. In the bank example, you can use **setf** with **account-name** and **account-balance**.

Readers and writers are called *accessor functions*.

The **statice:define-entity-type** form for **bank** also defines an *inverse reader function* called **account-named**, which is the inverse of **account-name**: given a name, it returns the account entity. (Inverse reader functions are not defined by default; you can request them by using the **:inverse** attribute option, as we did when we defined the bank entity type.)

deposit-to-account adds to someone's balance. First, **account-named** finds the account entity from the name provided. Then **account-balance** accesses and updates the balance of the account.

```
(defun deposit-to-account (name amount)
  (with-database (db *bank-pathname*)
    (with-transaction ()
      (incf (account-balance (account-named name)) amount))))
```

transfer-between-accounts moves a specified amount from one account into another account. If there are insufficient funds in the "from" account, it signals an error.

```
(defun transfer-between-accounts (from-name to-name amount)
  (with-database (db *bank-pathname*)
    (with-transaction ()
      (decf (account-balance (account-named from-name)) amount)
      (incf (account-balance (account-named to-name)) amount)
      (when (minusp (account-balance (account-named from-name)))
        (error "Insufficient funds in ~A's account" from-name)))))
```

See the section "Accessing Information in a Statice Database", page 14.

How to Access all Entities of an Entity Type

bank-total computes the total of all accounts in the database. It uses a special form called **statice:for-each**, which successively binds the variable **a** to each account entity.

```
(defun bank-total ()
  (with-database (db *bank-pathname*)
    (with-transaction ()
      (let ((result 0))
        (for-each ((a account))
          (incf result (account-balance a)))
        result))))
```

See the section "Iterating Over an Entity Type", page 16.

Benefits of Using Statice

We could have written this program using structures or instances to represent bank accounts, but by using Statice we gain two key advantages.

- *Persistence*. The account information is stored in a database and continues to exist even after the Lisp environment is destroyed by a cold-boot or system crash.

- *Sharing*. Any number of hosts on the network can use the database at the same time. Each one sees the effects of changes made by the others.

The use of transactions provides further benefits. Actions performed within a transaction are:

- *Atomic*: Either they all happen, or none of them happens. In **transfer-between-accounts**, either the **decf** and the **incf** both happen, or neither happens, and so the database cannot be left in an inconsistent state. If the error is signalled and not handled, the transaction *aborts*, and the **decf** and the **incf** are both undone. Transactions are always atomic even if the process is killed or the machine crashes.

- *Isolated*: Many processes can access the database at once, but any transaction always gets a consistent view of the database, as if there were no other processes. If you call **bank-total** while some other process is calling **transfer-between-accounts**, you'll get a correct total, because **bank-total** won't "see" the state in between the **decf** and the **incf**.

See the section "Introduction to Statice Transactions", page 10.

Entities and Entity Types

A Statice database holds a set of *entities*. Each entity represents some thing or concept in the real world. Every entity has a type, called its *entity type*. In the bank example, there is one entity type, named **account**.

The statice:define-schema Form

The first thing in the bank example is a **statice:define-schema** form.

```
(define-schema bank (account))
```

A *schema* is a description of everything that can appear in a database; this description consists of a list of the entity types. In the bank example, we define a schema named **bank**, and say that all the entities in the database are of entity type **account**. The symbol **bank** is called the *schema name*. In other words, a **bank** database contains **account** entities.

The statice:define-entity-type Form

The second thing in the bank example is a **statice:define-entity-type** form, which defines the **account** entity type.

```
(define-entity-type account ()
  ((name string :inverse account-named :unique t)
   (balance integer)))
```

An entity type can inherit from other entity types. The list following the entity type name includes those entity types that this one inherits from. In the bank example, **account** doesn't inherit from anything.

Attributes

Next is a list of descriptions of the *attributes* of the **account** entity type. Attributes are used to represent properties of entities and relationships between entities. Each attribute has a *name* and a *type*. In the bank example, there are two attributes. The attribute named **name** has type **string**, and the attribute named **balance** has type **integer**. This means that each account has a name, which is a string, and a balance, which is an integer. Attribute types are always *presentation types*, but only some presentation types are allowed. See the section "Statice Attribute Types", page 33.

Automatically-generated Functions

statice:define-entity-type automatically defines *accessor functions* for each attribute. In the bank example, there are two reader functions, **account-name** and **account-balance**. There are two corresponding writer functions, which you call by using **setf** with **account-name** and **account-balance**.

statice:define-entity-type also automatically defines an *entity constructor function*, used to create new entities of this entity type. In the bank example, the entity constructor function is **make-account**.

Attribute Options

After the name and type of an attribute comes a set of *attribute options*, expressed as alternating keywords and values. In the bank example, the attribute **name** has two options. The **:inverse** option defines an *inverse reader function* named **account-named**. The **:unique** option says that only one account in the database can have a particular name.

2.1.3. Making a Statice Database

After defining the schema, the bank example makes a database.

The Statice File System

Every Statice database is stored in its own file in a special kind of file system called a *Statice file system*. This **defvar** form defines a dynamic variable that holds the pathname of the particular database we are working with:

```
(defvar *bank-pathname* #p"beet:>finance>bank")
```

The value of the variable ***bank-pathname*** is a pathname to a file on a Statice file system; we call this kind of pathname a *database pathname*, because it indicates the location of a Statice database. The "host" component of the pathname is **beet**, but **beet** is the name of a Statice file system rather the name of a host.

The functions in the program use ***bank-pathname*** as an implicit argument to specify the database. Notice that each **statice:with-database** form uses this pathname to refer to the database.

If we were working with more than one bank, and each bank had its own database, we would change the value of the ***bank-pathname*** variable from time to time as we changed our attention from one bank to another. The program could have also been written by having each function take a pathname as an explicit argument.

File-System Objects in the Namespace

To find out where **beet** resides, Statice consults the namespace system, looking for a namespace object of type **file-system** named **beet**. This namespace object says what actual host to use, along with other information. It would be possible to move the **beet** file system from one host to another, using tapes or disk packs, without modifying our example program.

Database Pathnames

Pathnames are used to name databases within a Statice File System. The pathnames are hierarchical, with component names separated by ">" characters, as they are in LMFS. Unlike in LMFS, there are no file types or file versions, just file names. In the bank example, the directory is **>finance>** and the name is **bank**.

Many familiar Genera commands can be used with database pathnames, such as Show Directory, Create Directory, Rename File, and Delete File. Genera's Dired and File System Editor tools can also be used with database pathnames. Relative

pathnames and wildcard pathnames work the same way as for LMFS pathnames. However, it's not possible to open database pathnames, because they refer to Statice databases rather than files. So Copy File, Edit File, and Show File don't work on database pathnames.

For more information: See the section "Statice Database Pathnames", page 144.

Making the Database

Next, the bank example defines **make-bank-database**, which calls the Statice function **statice:make-database** to actually create a new database.

```
(defun make-bank-database ()
  (make-database *bank-pathname* 'bank))
```

statice:make-database takes as arguments the pathname and the name of the schema for the new database. It makes a new database and copies the schema into the database. The newly created database contains the schema, but no entities. That is, the database is set up so that it can hold entities of type **account**, but there aren't any accounts yet.

2.1.4. Accessing a Statice Database

Using statice:with-database

In the bank example, the form **statice:with-database** surrounds every reference to the database. **statice:with-database** does the following:

1. Determines which database should be opened, based on the *pathname*.

2. Opens the database, if it's not already open.

3. Binds the specified variable to the database instance, during the execution of the body.

4. Makes this database be the current database, during the execution of the body.

Opening a Database

The first time a database is used by a Lisp environment, Statice *opens* it. Once it has been opened, it stays open, and need not be opened again until Lisp is cold-booted. Opening happens automatically; the only thing you'll notice is a pause the first time **statice:with-database** is used. **statice:make-database** also opens the database it makes, so if you were to run the bank example, even the first usage of **statice:with-database** would not have to open the database. (There is no need to close a database.)

Binding the Variable

In the bank example, the Lisp variable **db** is bound to a Lisp object called a *database instance* that represents the database. Database instances are used as arguments to various Statice functions. They are needed only by programs that refer to two different databases at the same time. The bank example, like many real Statice applications, only uses one database, and so it never uses the **db** variable at all.

The Current Database

Whenever any Statice functions are used, there must be a *current database*. The current database is used as the default database by many Statice functions. This is why we don't need to use the **db** variable. **statice:with-database** binds the current database throughout the dynamic scope of its body. Dynamic scope means that if the body calls another function, the same database is still current while that function runs, so the called function doesn't need to use **statice:with-database**.

2.1.5. Introduction to Statice Transactions

Operations on Statice databases are grouped together into *transactions*. Before any operation can be performed on a database, a transaction must be begun. A transaction terminates either successfully or unsuccessfully. When a transaction terminates successfully, we say it *commits*. When a transaction terminates unsuccessfully, we say it *aborts*. (Opening a database is not considered an operation on the database, so it need not be done within a transaction.)

Using statice:with-transaction

In the bank example, the form **statice:with-transaction** surrounds every reference to the database. A transaction begins when the **statice:with-transaction** special form is entered. If the **statice:with-transaction** form returns to its caller, the transaction commits. If the **statice:with-transaction** form exits abnormally, due to a **throw** or a **return** through the **statice:with-transaction** form, the transaction aborts. Anything else that unwinds the stack, such as the killing of the process, also causes the transaction to abort.

statice:with-transaction has dynamic scope. The empty list in the **statice:with-transaction** form is for keyword options and values, which are rarely used.

Transactions are Atomic

The group of operations performed within a transaction are performed *atomically*. If the transaction commits, all of its effects take place; if the transaction aborts, none of them take place. For example, if a transaction begins, and executes some operations that modify values in the database, and then the transaction aborts, the modifications are undone and the database is left in its original state.

In the bank example, the benefits of atomic transactions can be seen in the **transfer-between-accounts** function. Because the two accounts are modified within

a single transaction, we can be sure that either the amount will be moved from one account to the other, or else nothing will happen. The total amount in all accounts is guaranteed to be stay the same; the database as a whole remains consistent.

In general, the atomic property of transactions prevents databases from being left in inconsistent, intermediate states. A transaction that modifies a database takes the database from one consistent state to another consistent state. Of course, the meaning of "consistent" depends on the application. In the bank example, consistency means that no amount enters or leaves the database due to a transfer between accounts.

Transactions are Isolated

Statice allows concurrent access to databases: more than one process can access a database at the same time. The processes might be on the same host or on different hosts. Transactions are used to keep these processes out of each other's way. The operations done in a transaction are *isolated* from all other transactions. This means that no transaction in progress is ever aware of the effects of another transaction in progress. In fact, transactions let Statice applications disregard concurrency altogether, so you can write programs as if the database were reserved for yourself.

In the bank example, suppose there were two processes. The first process is running the **transfer-between-accounts** function, in a transaction we'll call T1. The second process is running the **bank-total** function, in a transaction we'll call T2. Now, suppose the operations of these two transactions happened to occur in the following order:

1. T1 subtracts the amount from the from-account.

2. T2 iterates over all accounts, adding up the balances.

3. T1 adds the amount to the to-account.

4. Both transactions commit.

If this were allowed to happen, **bank-total** would return the wrong answer: it would be short by the amount being transferred. Transactions make sure that this cannot happen. The two transactions are isolated from each other, so T2 never observes the results of a transaction in progress. In this case, as soon as transaction T1 tried to modify the from-account, it would wait until transaction T2 terminates, and then proceed.

Transactions and System Failure

A *system failure* is any event that causes the entire system to stop, requiring a warm boot or cold boot. When a system failure occurs, Statice databases are not damaged. Any transactions that were in progress at the time of the failure are aborted, which means their effects are discarded. The database is left in a consis-

tent state. In other words, the transactions are atomic even if there is a system crash in the middle of a transaction.

When a database is being used over a network, there is a *server host* that actually stores the database, and one or many *client hosts* that run Statice programs affecting the database. (See the section "Using Statice Locally or Remotely", page 141.) A system failure on a user host aborts all transactions being done by that host. A system failure on a server host aborts all transactions being done by any user host that involve databases on this server. In any case, all unfinished transactions are aborted, and the database is left consistent.

When a transaction commits, the results of the transaction are written into the database, and will be visible to any future transactions. As soon as a **statice:with-transaction** form returns, Statice guarantees that the changes made by that transaction are persistently stored, and will be remembered even if there is a system failure.

Nested Transactions

If you nest a **statice:with-transaction** dynamically within another **statice:with-transaction** form, and the outer one is aborted, then both the outer transaction and the inner one are aborted. Nothing is committed until the end of the outermost **statice:with-transaction** is reached.

Errors and Transactions

If an error is signalled during a transaction, the normal Genera signalling mechanism begins: the lists of handlers are searched to find a handler for this error. The mere signalling of an error does *not* cause the transaction to abort. The transaction is aborted only if the error handler causes a throw to outside the scope of the **statice:with-transaction**.

If the error is not handled by any bound handler, default handler, or global handler, it causes the debugger to be invoked. If you then press `ABORT`, the debugger throws to the nearest restart handler, which is normally outside the scope of the **statice:with-transaction**. So, if an error reaches the debugger and you press `ABORT`, that normally aborts the transaction. In the bank example, if the "Insufficient funds" error is signalled, the debugger is entered, and the user will presumably abort the transaction.

But if the error is handled, the handler need not abort the transaction. For example, if there were a **condition-case** within the scope of a **statice:with-transaction**, and it handled a condition, the transaction would not be aborted.

2.1.6. Making New Statice Entities

Example of Making New Entities

The next function in the bank example is **make-new-account**.

```
(defun make-new-account (new-name new-balance)
  (with-database (db *bank-pathname*)
    (with-transaction ()
      (make-account :name new-name :balance new-balance))))
```

make-new-account takes arguments called **new-name** and **new-balance**, which should be a string and an integer, respectively. After opening the database and starting a transaction, **make-new-account** calls the function **make-account**.

Entity Constructor Functions

make-account is an *entity constructor* function. It makes a new entity of type **account** in the database. It also initializes the values of the **name** and **balance** attributes of the new entity to the values of the variables **new-name** and **new-balance**.

Entity constructor functions are defined automatically for each entity type. Their names are formed by prefixing the entity type name with **make-**. You can use the **:constructor** option to **statice:define-entity-type** to specify a different name for the constructor.

Constructors take keywords arguments, one for every attribute of the entity type. If a keyword is given with a value to the constructor, that value is stored as the value of the corresponding attribute. The names of the keywords are the same as the names of the attributes, but in the keyword package.

Entity Handles

The value returned by an entity constructor function is a Lisp object called an *entity handle*. An entity handle is an object in the Lisp world that represents an entity in a Statice database. Lisp programs pass around and manipulate entity handles in order to refer to entities.

An entity handle is an instance of a flavor. For every entity type, Statice defines a flavor named the same as the entity type. In our example, there is a flavor named **account**, and the entity handle returned by **make-account** is an instance of the **account** flavor. Remember that flavors are considered type names by the Lisp type system, and so the entity handle is of the Lisp type **account**, just as the corresponding entity is of the Statice entity type **account**.

Entity handles preserve the identity of entities. That is, there is never more than one entity handle in a Lisp world for a given entity. If you have two entity handles and want to know whether they refer to the same entity, you can use **eq** to check.

The entity type of an entity is fixed when the entity is created, and can never be changed. An entity stays the same type for its entire lifetime. An entity will not disappear from the database unless you explicitly delete it with **statice:delete-entity**.

2.1.7. Accessing Information in a Statice Database

Example of Accessing Information

The next function in the bank example is **deposit-to-account**. It adds a specified amount to the balance of the account with a specified name.

```
(defun deposit-to-account (name amount)
  (with-database (db *bank-pathname*)
    (with-transaction ()
      (incf (account-balance (account-named name)) amount))))
```

Accessor Functions: Readers and Writers

To get the value of an attribute of an entity, you use a *reader function*, or *reader*. To set the value of an attribute of an entity, you use **setf** with the reader. The function that is called when you use **setf** with a reader is called a *writer function*, or *writer*. Both readers and writers are called *accessor functions*, because they enable you to access the value of an attribute for either reading or writing.

Reader functions are defined automatically for every attribute of every entity type. The name of a reader is formed by concatenating the entity type name, a hyphen, and the attribute name. In the bank example, two readers are defined, **account-name** and **account-balance**.

Writers are also defined automatically. To call a writer, use the **setf** syntax with a reader, such as:

```
(setf (account-balance account) new-value)
```

statice:define-entity-type offers four options that enable you to specify the name of readers and writers: **:reader**, **:writer**, **:accessor**, and **:conc-name**. See the special form **statice:define-entity-type**, page 187.

Accessors are implemented as generic functions, which means you can write methods for them to specialize their behavior. For example, the reader **account-balance** is a generic function. Statice defines one method for it, on the **account** flavor. The argument to **account-balance** is an entity handle of type **account**. This reader returns the value of the **balance** attribute of the entity referred to by its entity handle argument. That is, **account-balance** takes an account and returns its balance, where the account is indicated by the entity handle for the account entity. The writer associated with **account-balance** is also a generic function with one method. Writers are implemented as **setf** generic functions. For information on defining methods for **setf** generic functions (also called setter functions): See the section "Setter and Locator Function Specs" in *Symbolics Common Lisp Programming Constructs*.

Inverse Reader Functions

The function **account-named** is an *inverse reader function*. Inverse readers are defined when you use the **:inverse** attribute option in the schema. You specify the name of the inverse reader; this is the value of the **:inverse** option.

account-named takes a string argument, and returns the entity handle for the **account** entity whose **name** value is the same as the argument. The operation performed by **account-named** is the inverse of the operations performed by **account-name**: the former goes from a string to the corresponding entity handle, while the latter does the opposite.

Kind of Function	*Argument*	*Value*
Inverse Reader	attribute's value	entity handle
Reader	entity handle	attribute's value

We discuss inverse writer functions later: See the section "Inverse Writer Functions for Entity-typed Attributes", page 26.

How deposit-to-account Works

Now we look at how the function **deposit-to-account** works. It first opens the database and starts a transaction. It calls **account-named** to find the account; **account-named** returns an entity handle that refers to the desired account entity. It calls **account-balance** to read out the current balance, adds in the specified amount, and then writes the sum back into **account-balance**. Finally, the body of **statice:with-transaction** returns, and the transaction commits. The results are now stored in the database.

Why Transaction Isolation is Important

deposit-to-account shows why it's important that transactions are isolated from each other. Suppose there is an account named "George" with a balance of 100. Suppose that there were two concurrent transactions, each trying to deposit 10 into George's account. If everything works properly, there should be 120 in George's account when the two transactions complete. But suppose they took place this way:

1. The first transaction reads the **account-balance**, and gets 100.

2. The second transaction reads the **account-balance**, and gets 100.

3. The first transaction adds 10 to the 100 that it read, and writes 110 into **account-balance**.

4. The second transaction adds 10 to its own 100, and writes 110 into **account-balance**.

5. Both transactions commit.

If this could happen, George's balance would only be 110 even though both transactions seemed to do their job. But in Statice this cannot happen because transactions are isolated from each other. Statice guarantees that if these two transactions are run concurrently, the overall effect will the the same as if they had run separately, one after the other. That's why **deposit-to-account** must do both its

reading and its writing within a single transaction.

How transfer-between-accounts Works

The **transfer-between-accounts** function does the same kinds of things as **deposit-to-account**. Again, it's important that all the operations be performed within a single transaction, to assure that concurrent transactions don't interfere with each other.

Another important reason to use a single transaction is that **transfer-between-accounts** does two semantically related operations that write into the database. We must be sure that either both operations take place, or neither. The transaction assures us that the two operations will be done atomically, even if the system crashes, the process is killed, the user types `c-m-ABORT`, or the error is signalled and leads to a throw.

2.1.8. Iterating Over an Entity Type

Example of Iteration

The final function in the bank example is **bank-total**, a function that returns the sum of the balances of all the accounts in the database.

```
(defun bank-total ()
  (with-database (db *bank-pathname*)
    (with-transaction ()
      (let ((result 0))
        (for-each ((a account))
          (incf result (account-balance a)))
        result))))
```

The statice:for-each Special Form

In order to add up the balances of all accounts in the database, we need a way to find all accounts in the database. The **statice:for-each** special form lets us do this. In the **bank-total** function, **statice:for-each** establishes a variable called **a**, and binds **a** successively to entity handles for each **account** entity in the database. It runs the body once for each entity, and the body accumulates the sum of the balances.

The extra level of list structure in the syntax of **statice:for-each**, is needed because **statice:for-each** has many other capabilities. It can iterate over a selected subset of entities; it can iterate in sorted orders; and it can iterate over tuples of entities from different entity types. Here we see **statice:for-each** in its most basic form, in which it iterates over all entities of a given entity type.

Databases Keep Track of All Entities

There is an important difference between Lisp objects and Statice entities. Lisp objects are kept track of only if you save references to them. It is possible for a Lisp object to be unreferenced and unreachable. Such an object is called "garbage" and can be deallocated automatically.

In contrast, Statice keeps track of all entities in a database. It can always access any entity, by using **statice:for-each** on the entity type of the entity. As a result, no entities ever become garbage.

Notice that **make-new-account** (defined in the section "Making New Statice Entities") makes a new entity, but never "puts" it anywhere. In Lisp, if you make a new Lisp object and then just ignore it, it immediately becomes garbage. But in Statice, it's installed into a database and can always be found again.

2.2. Using Statice for the First Time

When you are ready to begin using Statice, you need to do some preparation work before writing programs:

1. Find or create a Statice File System in which to store your database.

   First find out whether someone at your site has already created a Statice File System. If so, proceed to the next step. If not, you need to create one before proceeding further. See the section "Creating a New Statice File System", page 18.

   If you do not know whether or not there is already a Statice File System at your site, you can get that information by using the command Show All Statice File Systems. See the section "Show All Statice File Systems Command", page 170.

2. Create a directory in a Statice File System for your use.

   In general, several users will share one Statice File System. Users create their own directories within that Statice File System in which to keep their databases. Creating a directory in a Statice File System is just like creating a directory in any hierarchical file system. For example, if the Statice File System is named "beet" and you want to create a directory under the root named "finance", you could use this command:

   ```
   Create Directory beet:>finance>
   ```

3. Give the command: Add ASYNCH DBFS PAGE Service

   This command adds two service entries to your host object in the namespace database. Once you've run this command on a particular host, you never need to run it again. (If you do run it again, it won't hurt.) Each client machine that uses Statice should have these service entries.

4. Decide what package to write your Statice program in.

   Now you're ready to write programs using Statice. For any Statice application program, you should make your own package that uses the **statice** package. See the section "Writing Statice Programs in the Right Package", page 20.

2.2.1. Creating a New Statice File System

If you are the first person to use Statice at your site, you need to create a Statice File System. A Statice File System is a file system that contains Statice databases.

1. Decide which host the Statice File System will reside on.

   A Statice File System must reside on a Symbolics host, and cannot reside on any other kind of host. If you only have one Symbolics host, of course, your decision is simple. Otherwise, it depends on how your site is managed.

   At sites with many Symbolics hosts, usually one or more machines act as dedicated server hosts. Usually, these hosts spend all their time providing services to users on other hosts, and they often have shared facilities such as tape drives, and lot of disk space. This is the ideal place to put a Statice File System.

   In some cases, you might want to put the Statice File System on your own host. If your site has no dedicated server, or if the server is running a Genera release previous to 7.2, you cannot put the Statice File System on the server. If you are trying out Statice for the first time, you might not want to use the server until you feel more confident.

   We recommend that any Statice File System that is going to hold real data (as opposed to a Statice File System used only for experimentation) should be on a host with its own tape drive. It's important to do regular backups to tape, to guard against the possibility of disk failure. You can do backups to a tape drive on another host by using remote tape access, but using a tape drive on the same host as the Statice File System is faster and more reliable.

2. Make disk space available on the machine that will store the Statice File System.

   A Statice File System lives in the FEP file system, just like everything else that occupies disk space on a Symbolics host. For general information about the FEP file system: See the section "FEP File Systems" in *Site Operations*.

   A Statice File System occupies some disk space, so you'll need some free space in the FEP file system of one or more disks on the host. You can use the Show FEP Directory command to find out how much space is available on each disk, and find candidates to be deleted and expunged if you need to make more room.

3. Choose a name for the Statice File System.

   This name will be the name of the file system object in the namespace database. Just be sure to pick a name that has not already been used. You

can use the Show All Statice File Systems command to see all the names that are already in use. More important, pick a name that is not a host name at your site, so that the pathname parser can distinguish the pathnames for your file system from the pathnames for hosts at your site.

4. Create the Statice File System.

   On the machine you have designated to store the Statice File System, use the Create Statice File System command. This command requires the name of the Statice File System as its first argument.

   ```
   Create Statice File System name
   ```

   An AVV menu is then displayed, showing you the amount of free space on each of the disks. The variables appear something like this:

   Directory partition: FEP1:>Statice>*name*.UFD
   Maximum directory entries: 1000
   Maximum log size in blocks: 500
   Partition: FEP1:>Statice>*name*-part0.file.newest
   Blocks (**None** to remove): **None** *integer*

   The UFD extension to the filename is used for the directory partition of a Statice file system. Usually, the defaults for the first four variables in this menu are fine, and you need only enter a value for the fifth. The Blocks variable indicates the size (how many blocks) the partition should be. A Statice File System can span several different partitions, or it can use only one. This enables you to have a Statice File System that uses space on several disks.

   If you want the partition to be 5000 blocks long, click on *integer* and enter 5000. The AVV menu will change to offer you a second partition. If you want a second partition, you can enter a size in blocks, and a third partition will be offered, and so on. Otherwise, just leave it alone; if the size is given as **None**, no partition is created.

   The log is a special file used by Statice to implement recovery from failures. Its default initial size is 500 blocks. If you want to make a very small Statice file system, you can change this number to be smaller. The log will grow automatically when necessary.

   When you are satisfied with the values of the variables, press `END`. A name-space object is created, of type **file-system** and of name *name*. The Statice File System is created on the machine, including the FEP files that hold the directory partition and all of the file partitions you specified. It also allocates the log file, with the specified size.

5. Configure the server machine by giving the Add DBFS PAGE Service command.

If you want this machine to be usable as a server, give the Add DBFS PAGE Service command. This command adds two service entries to the host object in the namespace database, allowing other hosts to use this host as a server. Once you have given this command on a particular host, you never need to run it again. (If you do run it again, it won't hurt.) If you do not want this machine to be usable as a server, omit this step.

You've now created a Statice File System. Before you start using Statice: See the section "Writing Statice Programs in the Right Package", page 20.

For detailed information about Statice administration: See the section "The Architecture of Statice", page 141.

2.2.2. Writing Statice Programs in the Right Package

When developing a program that uses Statice, you should not put your own file directly in the **statice** package; doing so would create name conflicts with internal Statice functions.

For any Statice application program, you should make your own package that uses the **statice** package. You can do this via the **Package:** entry in the file attribute list. You can also declare the package using **defpackage** or **make-package**. By creating your own package, you avoid conflicts with other programs that might be loaded with yours.

If you are just experimenting with Statice and don't want to make a new package, you can write your program in the **statice-user** package, or **s-u** for short. **statice-user** has no symbols of its own, but inherits symbols from **statice** and **scl**.

2.3. A More Complicated Schema: the University Example

2.3.1. Defining a Schema for a University

This part of the Statice tutorial uses a university example to introduce concepts related to the schema of a database. You can find the complete source listing for the university example in SYS:STATICE;EXAMPLES;UNIVERSITY-EXAMPLE.LISP.

We present the whole schema here. Just skim over it for now. We'll focus on the new features in subsequent sections.

```
;;; -*- Mode: LISP; Syntax: Common-lisp;
;;;     Package: (UNIVERSITY :USE (STATICE SCL));
;;;     Base: 10; Lowercase: Yes -*-

(define-schema university (person student graduate-student shirt
                           course instructor department))
```

```
(define-entity-type person ()
  ((name string :unique t :no-nulls t :cached t
         :inverse person-named :inverse-index t)
   (id-number integer :unique t :read-only t)))

(define-entity-type student (person)
  ((dept department :inverse students-in-dept)
   (courses (set-of course) :index t :inverse course-students)
   (shirts (set-of shirt) :unique t :inverse shirt-owner)))

(define-entity-type graduate-student (student)
  ((thesis-advisor instructor))
  (:conc-name student-))

(define-entity-type shirt ()
  ((size integer)
   (color string :initform "white")
   (washed boolean)))

(define-entity-type course ()
  ((title string :inverse courses-entitled)
   (dept department)
   (instructor instructor))
  (:multiple-index (title dept) :unique t))

(define-entity-type instructor (person)
  ((rank rank :initform "Assistant")
   (dept department :no-nulls t)
   (visiting boolean)
   (salary single-float)))

(define-entity-type department ()
  ((name string :unique t
         :inverse department-named :inverse-index t)
   (head instructor)))

(define-presentation-type rank ()
   :expander '(dw:member-sequence ("Assistant" "Associate" "Full")))
```

2.3.2. Entity-Typed Attributes

An Entity-typed Attribute: dept

Look at the description of the **dept** attribute of the **student** entity type.

```
(define-entity-type student (person)
  ((dept department :inverse students-in-dept)
   (courses (set-of course) :index t :inverse course-students)
   (shirts (set-of shirt) :unique t :inverse shirt-owner)))
```

The type of the **dept** attribute is **department**, which is a Statice entity type. This means that the value of the **dept** attribute for any **student** entity is a **department** entity.

In English, this means that every student has an associated department, presumably the department in which the student is majoring. Students and departments are both represented by entities in Statice, and the **dept** attribute serves to link one entity to another. Statice attributes can serve as relationships between entities as well as properties of entities.

Entity Types Versus Ordinary Types

Every Statice attribute has a type, which is either an *ordinary type* or an *entity type*. An *entity type* is defined by **statice:define-entity-type**, and the values of the attribute are always entities of that type. An *ordinary type* is something like **integer** or **string**. Statice pre-defines many useful ordinary types. It is also possible to define your own ordinary type, with **statice-type:define-value-type**; this advanced feature is described later: See the section "Defining New Statice Types", page 84.

The type of an attribute is always specified by a presentation type. This is also true of entity-typed attributes: every entity type is a presentation type. This is because defining an entity type defines a flavor of the same name, and flavor names are automatically type names.

2.3.3. Set-Valued Attributes

Example of a Set-valued Attribute

Look at the description of the **courses** attribute of the **student** entity type:

```
(define-entity-type student (person)
  ((dept department :inverse students-in-dept)
   (courses (set-of course) :index t :inverse course-students)
   (shirts (set-of shirt) :unique t :inverse shirt-owner)))
```

Like **dept**, **courses** is an entity-typed attribute. But the presence of the special symbol **set-of** means that it is also a *set-valued* attribute: the attribute holds a set of values rather than a single value. In this case, every student is taking a set of courses, and each course is represented by a Statice entity of type **course**.

Every Statice attribute is either single-valued or set-valued. In the bank example (defined in the section "Quick Overview of Statice: the Bank Example"), both attributes were single-valued. The **university** schema, like most real schemas, has several set-valued attributes.

We usually give set-valued attributes plural names, to make it clear that they hold many things.

Accessor and Entity Constructor Functions

When you use the **student-courses** reader function, it returns a list of entity handles, one for each course that the student is taking. You can also set the **courses** of a student by using **setf** with **student-courses**, and giving it a new list of entities.

When you use the **make-student** entity constructor function, the value you provide with the **:courses** keyword must be a list of entity handles (possibly empty, of course). If you don't provide any initialization, the initial value of a set-valued attribute is the empty set.

Using statice:add-to-set and statice:delete-from-set

Two special forms are provided for adding an element to a set, or deleting an element from a set. If the value of **joe-cool** is a student (an entity handle of type **student**, that is), and the value of **english-101** is a course, the following form adds this course to Joe's set of courses:

```
(add-to-set (student-courses joe-cool) english-101))
```

The following form deletes this course from Joe's set of courses:

```
(delete-from-set (student-courses joe-cool) english-101))
```

In these forms, **english-101** is a Lisp form that is evaluated to return a course, and **joe-cool** is a Lisp form that is evaluated to produce a student, but **(student-courses joe-cool)** is not evaluated; it simply identifies the set.

2.3.4. Iterating Over Sets with statice:for-each

Iterating Over All Entities of an Entity Type

The function **mean-salary-of-instructors** returns the mean of the salary of all the instructors.

```
(defun mean-salary-of-instructors ()
  (with-database (db *university-pathname*)
    (with-transaction ()
      (let ((total-salary 0.0)
            (number-of-instructors 0))
        (for-each ((i instructor))
          (incf total-salary (instructor-salary i))
          (incf number-of-instructors))
        (/ total-salary number-of-instructors)))))
```

The special form **statice:for-each** is an iteration construct, like **dotimes**. In **mean-salary-of-instructors**, **statice:for-each** evaluates its body once for each **instructor** entity in the database. Each time around the loop, the variable **i** is bound to the entity handle of the next **instructor** entity. Inside the body, the reader function **instructor-salary** obtains the salary of the instructor under consideration, and this amount is added into the variable **total-salary**.

The general form for iterating over all entities of a type is:

```
(for-each ((variable entity-type-name))
  body...)
```

variable is a symbol, which names a new variable, just as in **dotimes**. *entity-type-name* is a symbol, which must be the name of an entity type in the current database.

Iterating Over Members of a Set-valued Attribute

The function **all-shirts-washed** finds all the shirts of a particular student, and marks them as being washed. The argument must be an entity handle of type **student**.

```
(defun all-shirts-washed (clean-student)
  (with-database (db *university-pathname*)
    (with-transaction ()
      (for-each ((s (student-shirts clean-student)))
        (setf (shirt-washed s) t)))))
```

(student-shirts clean-student) has the syntax of a call to a reader function. It refers to the set of all shirts owned by this particular student. **statice:for-each** evaluates its body once for each member of this set, binding **s** to the entity handle for each **shirt** entity in the set.

The general form for iterating over all members of a set is:

```
(for-each ((variable (reader-function-name form)))
  body...)
```

variable is a symbol, the same as above. *reader-function-name* is the name of the reader function of an attribute. The attribute must be set-valued and entity-typed. The attribute's entity type must be in the current database. *form* can be any Lisp form. It is evaluated, once, before the iteration begins. The value of *form* must be an entity of the attribute's entity type. **statice:for-each** iterates over the members of the attribute (specified by *reader-function-name*) of the entity (specified by the value of *form*).

2.3.5. One-to-One, Many-to-One, and Other Relationships

Every Statice reader function, including the inverse ones, can be characterized as *one-to-one*, *many-to-one*, *one-to-many*, or *many-to-many*. This is best explained by example.

```
(define-entity-type person ()
  ((name string :unique t :no-nulls t :cached t
       :inverse person-named :inverse-index t)
   (id-number integer :unique t :read-only t)))

(define-entity-type student (person)
  ((dept department :inverse students-in-dept)
   (courses (set-of course) :index t :inverse course-students)
   (shirts (set-of shirt) :unique t :inverse shirt-owner)))
```

- **person-name** is a one-to-one function. Every person has one name, and every name refers to just one person. There is one person to one name.

- **student-dept** is a many-to-one function. Every department has many students in it, but every student has one particular department. There are many students to one department.

- **student-shirts** is a one-to-many function. Every shirt is owned by just one student, but every student has many shirts. There is one student for many shirts.

- **student-courses** is a many-to-many function. Each student takes many courses, and each course is taken by many students. There are many students to many courses.

The inverse functions work just the same way.

- **person-named** is a one-to-one function. Every name refers to just one person, and and every person has one name. There is one name to one person.

- **shirt-owner** is a many-to-one function. Every student is the owner of many shirts, but every shirt is owned by just one student. There are many shirts to one owner.

- **students-in-dept** is a one-to-many function. Every student has only one particular department, but every department has many students in it. There is one department to many students.

- **course-students** is a many-to-many function. Each course is taken by many students, and each student takes many courses. There are many courses to many students.

The General Rule

What determines each function's characterization into one of these four kinds? For regular reader functions, if the function is set-valued, it's a something-to-many function, and if the function is unique, it's a one-to-something function. The inverse reader function (if any) always has the inverse characterization; for example, if an attribute's reader function is one-to-many, the inverse reader function is many-to-one.

Here are all the possibilities:

Unique	*Set-valued*	*Reader*	*Inverse reader*
Yes	No	One-to-one	One-to-one
No	No	Many-to-one	One-to-many
Yes	Yes	One-to-many	Many-to-one
No	Yes	Many-to-Many	Many-to-Many

Designing the Schema

When you're designing a schema, be careful to think about each attribute, and decide which kind of reader function you want it to have.

The designer of the **university** schema had to make several important choices. For example, **person-name** is one-to-one. This is convenient, because we can speak of "the person named Fred Smith". However, if a second Fred Smith shows up at the university, it's going to be a problem.

The **title** attribute of **course** is not unique, though. **course-title** is a many-to-one function. That's because two different departments sometimes pick the same course name. To specify a particular course at the university, you need both its title and its department.

The schema could be made more flexible. Instructors might not have only one department; some might divide their time among several departments. A department could have two people who share being the head of the department. However, it's only worth modelling such circumstances if you intend to write all your programs to handle all these possibilities. The extra flexibility costs in complexity and performance. Designing a schema takes care and judgement.

2.3.6. Inverse Writer Functions for Entity-typed Attributes

Inverse Writer Functions

If you use the **:inverse** option to **statice:define-entity-type** to define an inverse reader function, and the attribute is an entity-typed attribute, Statice defines an *inverse writer function* as well as an inverse reader. To call the inverse writer, use **setf** with the inverse reader.

An entity-typed attribute defines a relationship between two entities:

entity-1 The entity that defines the attribute

entity-2 The entity that is the value of the attribute

The inverse reader enables you to go from *entity-2* to get *entity-1*. The inverse writer takes *entity-2* as an argument, and gives a new value, so that the next time you use the inverse reader on *entity-2*, the result is the new value. The new value itself is an entity, which might be *entity-3*. Thus, the effect of using the inverse reader is to change *entity-3* such that its attribute is given the value *entity-2*.

Inverse Writers and Set-valued Attributes

The above discussion is simplified because we assume that the entity-typed attribute is unique, so it defines a one-to-one relationship between two entities. In fact, you can use an inverse writer even for a non-unique attribute (as long as it is entity-typed and you specified the **:inverse** option). The inverse reader for a non-unique attribute is set-valued, so the inverse writer changes a set of entities.

Consider the **dept** attribute of the **student** entity type:

```
(define-entity-type student (person)
  ((dept department :inverse students-in-dept)
   (courses (set-of course) :index t :inverse course-students)
   (shirts (set-of shirt) :unique t :inverse shirt-owner)))
```

The **dept** attribute is entity-valued, and the **:inverse** option is used to define the inverse reader **students-in-dept**. Thus, you can use **setf** with **students-in-dept**. Since **dept** is not unique, the inverse reader **students-in-dept** is set-valued. Thus, the new value must be a list of students. The effect is to change all affected student entities:

- For all students listed in the new value given to the inverse writer: the students' **dept** attribute is made to be the specified department.

- For any students not list in the new value given to the inverse writer, but whose **dept** was previously the specified department: these students' **dept** attribute is made to be the null value. (If **:no-nulls** were specified for the **dept** attribute, an error would be signaled.)

No Inverse Writers for Some Attributes

An inverse writer is defined only for entity-typed attributes (when the **:inverse** option is used). There is no inverse writer for attributes whose types are ordinary types, such as strings, integers, and so on.

Consider the **name** attribute of the **person** entity type:

```
(define-entity-type person ()
  ((name string :unique t :no-nulls t :cached t
        :inverse person-named :inverse-index t)
   (id-number integer :unique t :read-only t)))
```

Remember that an inverse reader goes from the value of an attribute to the entity. Thus, you can use **person-named** to get the person entity whose name is a given string. An inverse writer changes the value of the attribute; if there were a inverse writer in this case, it would change the person named "Joe" to being another person, who already has a name of his own. The semantics of such an operation are unclear. If the goal is to change one person's name to being a different string, the straightforward approach is to use a normal writer function on that person entity (rather than using an inverse writer on a string). Due to semantic ambiguities, Statice does not provide inverse writers for attibutes whose types are ordinary types.

2.3.7. Inheritance From Entity Types

An entity type can *inherit* from other entity types, just as flavors can inherit from other flavors. Look at the entity types **person**, **student**, and **graduate-student** in the **university** schema.

```
(define-entity-type person ()
  ((name string :unique t :no-nulls t :cached t
       :inverse person-named :inverse-index t)
   (id-number integer :unique t :read-only t)))

(define-entity-type student (person)
  ((dept department :inverse students-in-dept)
   (courses (set-of course) :index t :inverse course-students)
   (shirts (set-of shirt) :unique t :inverse shirt-owner)))

(define-entity-type graduate-student (student)
  ((thesis-advisor instructor))
  (:conc-name student-))
```

student inherits from **person**, and **graduate-student** inherits from **student**. This is because a student is a kind of person, and a graduate student is a kind of student.

Inheritance of Attributes

Since a student is a kind of person, and each person has a name, it follows that a student has a name. We say that entity type **student** inherits the **name** attribute from entity type **person**. In general, entity types inherit all of their ancestors' attributes.

The reader functions defined by the ancestor entity types can be used on entities of the descendant entity types. For example, the reader function **person-name** can be applied to entities of type **student**, or of type **graduate-student**. Note that there is no reader function named **student-name**—each attribute has only one reader function.

The uniqueness of the **name** attribute applies to every entity that has the attribute, no matter what its entity type. That is, you can't have a **person** entity with name **"Fred"** and also have a **student** entity with name **"Fred"**. Every value of the **name** attribute, in every entity in which the **name** attribute exists, must have a distinct value. The inverse reader function **person-named** can return **person** entities, **student** entities, or **graduate-student** entities.

Inheritance of Types

Type membership is affected by inheritance the same way as in Flavors. Every entity of type **graduate-student** is also considered to be of type **student**, and every entity of type **student** is also considered to be of type **person**.

For example, the argument of the **person-name** is required to be of type **person**. It's OK to call **person-name** on a **student** entity, because such an entity is a **person** too.

Another example: if there were an attribute whose type were **student**, it would be all right for the value of the attribute in some entity to be a **graduate-student** entity, since every graduate student is a student. But it would not be all right for the value to be a **person**, since not every person is a student.

Duplicate Attribute Names are Not Allowed

Statice does not allow an entity to specify an attribute name which is the same as an attribute name of one of its parent entity types.

This restriction is one difference between Statice and Flavors (and CLOS). A flavor can specify the name of an instance variable even if it is the name of an inherited instance variable, with the goal of modifying it by giving it a new option, or an option to override an inherited one.

2.3.8. The Statice Null Value

Statice has a special value called the *null value*. The value of any single-valued attribute can be the the null value (unless **:no-nulls** is used), regardless of the type of the attribute. The null value is used to represent "value unknown" or "not applicable".

For example, the value of the **id-number** attribute for a particular person might be the null value. This would represent that the person does not have any assigned ID number. For brevity, we often say that the value of the attribute "is null".

The null value is not the same as an empty string, or zero. If the value of a person's **id-number** attribute is zero, the person *does* have an ID number, which just happens to be zero. The null value means that the person does not have any ID number at all.

The null value *never* appears in set-valued attributes, only in single-valued attributes.

Reader Functions and the Null Value

What happens if you call a reader function, and the value of the attribute is the null value? Reader functions actually return two values. For the reader of a single-valued attribute: If the value of the attribute is not null, the first value of the reader is the Lisp representation of the value, and the second value is **t**. If the value of the attribute is null, both returned values are **nil**. (For the reader of a set-valued attribute: the second value is always **t**.)

In general, if you want to know whether the value of an attribute is null, you have to receive and test the second value. For many types, though, **nil** isn't the Lisp representation of any non-null value, and so you can simply see whether the first returned value is **nil**. This works for attributes of type **integer** and **string**; it also works for entity-type attributes. It does not work for **boolean**, because **nil** is a possible value of a **boolean** attribute.

For example, if George has ID number 123 and Fred has no ID number:

```
(person-id-number george) => 123 t
(person-id-number fred)   => nil nil
```

If Prof. Smith is not a visiting instructor, but we don't know whether Prof. Jones is a visiting instructor:

```
(instructor-visiting prof-smith) => nil t
(instructor-visiting prof-jones) => nil nil
```

How do you set the value of an attribute to the null value? If the type of the attribute doesn't use **nil** as the Lisp representation of any non-null value, you can just use **setf** with **nil**. But for types such as **boolean**, setting to **nil** means setting to "false", so this doesn't work. The general way to set an attribute to the null value is the **statice:set-attribute-value-to-null** function.

For example, to remove George's ID number, and give Fred ID number 456:

```
(setf (person-id-number george) nil)
(setf (person-id-number fred) 456)
```

Since ID numbers are always integers, **nil** always means the null value. The second value returned by the reader is not needed, and we can make the value be null by simply storing **nil**.

To assert that we no longer know whether Prof. Smith is a visiting instructor, but we know that Prof. Jones is not:

```
(set-attribute-value-to-null prof-smith 'instructor-visiting)
(setf (instructor-visiting prof-jones) nil)
```

Since the **visiting** attribute is boolean, **nil** means false rather than the null value. So we must use the second value returned by the reader function to determine whether the value is null, and we must use **statice:set-attribute-value-to-null** to make the value be null.

Entity Constructors and the Null Value

When you make a new entity using an entity constructor function, there are two ways to make the initial value of a single-valued attribute be null.

First, if you simply omit the keyword for the attribute (and if **:initform** isn't used), the attribute starts out null. For example, the following form makes a person named "Beth", with a null ID number:

```
(make-person :name "Beth")
```

Second, if the type of the attribute doesn't use **nil** as the Lisp representation of any non-null value, you can supply **nil** as the initial value to mean the null value. For example, the following form makes a person named "Beth", with a null ID number, just like the last example:

```
(make-person :name "Beth" :id-number nil)
```

If the type of the attribute does use **nil** to represent some non-null value, you can only use the first way, not the second. That is, you have to omit the keyword for the attribute in order to initialize it to the null value.

If an attribute is set-valued, and you omit its keyword or provide **nil** as its value, the initial value is the empty set.

Null Values and the :unique Attribute Option

As we discussed earlier (in sections "Defining a Statice Schema" and "One-to-One, Many-to-One, and Other Relationships"), if an attribute has the **:unique** attribute option, no two entities can have the same value for that attribute. However, for purposes of **:unique** checking, one null value *does not equal* another null value. In our example, more than one person can have a null ID number, even though **id-number** is a **:unique** attribute.

This makes sense when you consider what the null value represents. The **:unique** attribute ensures that we don't assign the same ID number to two different people. But it is valid for several people not to have any ID number at all.

2.3.9. The :no-nulls Attribute Option

The **:no-nulls** attribute option is used in the **name** attribute of **person**:

```
(define-entity-type person ()
  ((name string :unique t :no-nulls t :cached t
         :inverse person-named :inverse-index t)
   (id-number integer :unique t :read-only t)))
```

:no-nulls means that values of the **name** attribute are not allowed to be the null value. **name** is a "required" attribute; every person must have some name. **:no-nulls** has the following consequences:

- The second value returned by the reader function for a **:no-nulls** attribute is always **t**.
- Any attempt to set the value of a **:no-nulls** attribute to the null value signals an error.
- When an entity creator function is called, values must be supplied for all **:no-nulls** attributes, either explicitly or with **:initform**.

Examples:

```
(setf (person-name george) nil) => error
(make-person :id-number 123) => error
```

Using :no-nulls in a Schema

In the **university** schema, the **:no-nulls** attribute is used sparingly. The designer of the schema allows for null values in many attributes. For example, a graduate student might enroll in the university, but not get a thesis advisor assigned for a while. We might want to represent an instructor in the database even if we don't yet know what the instructor's salary is. We might even want to represent a course whose title is still under dispute.

The **dept** attribute of **instructor** has the **:no-nulls** attribute. The designer of the schema asserts that whenever any instructor is represented in the database, the instructor must have some particular department.

Consider the quandry we'd be in if we used the **:no-nulls** attribute on the **head** attribute of **department**. Suppose we want to represent Prof. Einstein, who is the head of the Physics department, and is in the Physics department. No matter which one we create first, an error is signalled, because each demands the existence of the other. In fact, it would be impossible to create any departments or any instructors! The problem arises because of a *cycle* of **:no-nulls** attributes. Be careful not to make any.

2.3.10. The :initform Attribute Option

The **:initform** attribute option is used in the **color** attribute of **shirt**:

```
(define-entity-type shirt ()
  ((size integer)
   (color string :initform "white")
   (washed boolean)))
```

:initform means that if an entity is created, and the keyword for this attribute is not supplied, the value of the **:initform** option should be evaluated, and the result used as the initial value of the attribute. In other words, it's the default initial value for the attribute.

:initform can be used with single-valued or set-valued attributes. If the attribute is set-valued, the result of evaluating the option must be a list. **nil**, of course, is a list, representing the empty set.

Here are some examples.

```
(shirt-color (make-shirt :color "black")) => "black"
(shirt-color (make-shirt)) => "white"
(shirt-color (make-shirt :color nil)) => nil
```

In the last example, the value of the **color** attribute is the null value. Because the **:color** keyword was explicitly provided to the entity constructor function, the **:color nil** takes precedence over the **:initform**.

The value of the **:initform** attribute option doesn't have to be a constant. In fact, it can be an arbitrary Lisp form. The form is evaluated inside the lexical environment of the containing **statice:define-entity-type** form (this only matters if the **statice:define-entity-type** isn't at top level, which is uncommmon).

Summary of Rules for Initial Values

These rules specify how Statice determines the initial value of each attribute when the entity constructor function is called.

1. If the attribute's keyword is supplied to the entity constructor function, initialize the attribute to the associated value.

2. Otherwise, if the attribute has an **:initform** option, evaluate the **:initform** and initialize the attribute to the returned value.

3. Otherwise:

- If the attribute is single-valued, initialize the attribute to the null value.
- If the attribute is set-valued, initialize the attribute to the empty set.

These points should also be kept in mind for single-valued attributes:

- If **nil** is not a meaningful value for an attribute's type, **nil** can be used to mean the null value.
- If you try to initialize an attribute to the null value and **:no-nulls** is specified for that attribute, an error is signalled.

2.3.11. The :read-only Attribute Option

The **:read-only** attribute option is used in the **id-number** attribute of **person**:

```
(define-entity-type person ()
  ((name string :unique t :no-nulls t :cached t
         :inverse person-named :inverse-index t)
   (id-number integer :unique t :read-only t)))
```

:read-only means that no writer is defined for setting the value of the attribute. That is, you cannot use **setf** with the reader. Once a **person** entity has been created, the value of the **id-number** attribute for the person never changes.

For set-valued attributes, **:read-only** also means that the special forms **statice:add-to-set** and **statice:delete-from-set** cannot be used.

2.3.12. Statice Attribute Types

So far, most of the attributes we've examined were entity-typed, or of type **integer** or **string**. Statice provides many other types. The attributes of **instructor** use some other types:

```
(define-entity-type instructor (person)
  ((rank rank :initform "Assistant")
   (dept department :no-nulls t)
   (visiting boolean)
   (salary single-float))
  (:multiple-index (rank salary)))
```

The type of the **salary** attribute is **single-float**. Values of this attribute are floating-point numbers in IEEE-standard 32-bit format. Many other Lisp types can be used as attribute types, including the following. This is only a brief survey, for complete information: See the section "Built-In Statice Types", page 75.

Type	*Values*
integer	Arbitrary-precision integer.
(integer *n m*)	Integer subrange, as defined by Common Lisp.
string	Arbitrary-length string. Characters can have character styles.

(limited-string *n***)** String of *n* or fewer characters. Characters cannot have character styles or modifier bits and must be in the standard character set.

boolean **t** or **nil**.

single-float Floating point number in IEEE Single format.

double-float Floating point number in IEEE Double format.

number Any number.

symbol Lisp symbol, from any package. Characters in the print name can have character styles.

entity-handle Any entity, regardless of its entity type.

t Any Lisp object that can be handled by the binary file dumper and loader.

(member *a b c ...***)** Any of the Lisp objects *a*, *b*, or *c*. Compared using **eql**.

(alist-member :alist (("*a*" **.** *x***) (**"*b*" **.** *y***) (**"*c*" **.** *z***)))**
Any of the objects *x*, *y*, or *z*. The full syntax of the **alist-member** is also supported. Compared with **equalp**.

(dw:member-sequence (*a b c***))**
Any of the Lisp objects *a*, *b*, or *c*. Comparison predicate specified by **:test** data argument, defaulting to **eql**.

time:universal-time
Arbitrary-precision integer.

time:time-interval Arbitrary-precision integer.

time:time-interval-60ths
Arbitrary-precision integer.

Type Inheritance

Statice types are tied into the inheritance structure of presentation types. If a particular presentation type is not directly handled by Statice, we look at its ancestors to find one that is.

For example, **rational** can be used as a Statice type, because **number** is one of its ancestors in the inheritance structure. The writer function makes sure that only **rational** values can be stored, but the same storage format is used as for any **number**.

The t Type

The **t** type accepts any Lisp object. It uses the same binary format that the Genera Lisp compiler uses to produce its binary output files. The same data that would go into a binary file is, instead, stored in the Statice database. When the value is retrieved, the binary loader is called in. This is the same binary loader used by the **load** function and the :Load File command.

The semantics of **statice:lisp-object** are much like the semantics of printed representation. To store a Lisp object in a **statice:lisp-object** attribute value is like printing it, and to get a Lisp object from a **statice:lisp-object** attribute value is like reading it. If you store a list, the elements of the list are stored, recursively. If you get a list from the database, a *new list* is created.

For example, if **student-symbols** were a single-valued reader function of a **statice:lisp-object** attribute:

```
(setq x (list 'a 'b 'c))
(setf (student-symbols george) x)
(setq y (student-symbols george)) => (a b c)
(equal x y) => t
(eq x y) => nil
```

Enumerated Types

The types **member**, **alist-member**, and **dw:member-sequence** can all be used as *enumerated* types. An enumerated type has a small, fixed set of possible values.

For example, suppose we want to model cards in a playing deck:

```
(define-entity-type card ()
  ((number (integer 0 (13)))
   (suit (member spades hearts diamonds clubs))))

(setq ace-of-spaces (make-card :number 1 :suit 'spades))
(card-suit ace-of-spaces) => spades
```

member uses **eql** to determine equality. If the elements of the enumerated set are strings, the **alist-member** and **dw:member-sequence** types are more useful. For information on these types: See the section "Dictionary of Predefined Presentation Types" in *User Interface Dictionary*.

The Rank Type

In the **university** schema, the **rank** attribute of the **instructor** entity type has type **rank**. **rank** is a presentation type defined by the **university** program itself. **rank** can be used as a Statice type, even though it's not one of the built-in types, because of presentation type inheritance.

```
(define-presentation-type rank ()
  :expander '(dw:member-sequence ("Assistant" "Associate" "Full")))
```

rank has an **:expander** that expands into the **dw:member-sequence** presentation type, and **dw:member-sequence** is supported by Statice. The value of the **rank** attribute is always one of the strings **"Assistant"**, **"Associate"**, or **"Full"** (or the null value).

Physical Space Usage

Here we briefly discuss the amount of physical space required by values of some of the commonly used types. For more detailed information: See the section "Built-In Statice Types", page 75.

- Although **integer** can represent arbitrary precision, it only uses up one word for numbers whose absolute value is less than **(expt 2 30)**, which is about one billion.
- **boolean** values are packed into one bit if **:no-nulls** is used, and two bits if nulls are allowed.
- Integer subranges are packed into as few bits as are needed to represent the possible values, including nulls if they are allowed.
- Enumerated types are packed just like integer subranges.
- Strings have one word of overhead and use an integer number of words. A thin string *n* characters long uses **(1+ (ceiling** *n* **4))** words; fat strings (using character styles and/or character sets) take more space.

2.3.13. The :conc-name Entity Type Option

Entity types can have options. The options follow the list of attribute descriptions. Each is specified by a list that starts with the name of the option. The name is a keyword. For example, the **graduate-student** entity type has one option, **:conc-name**.

```
(define-entity-type graduate-student (student)
  ((thesis-advisor instructor))
  (:conc-name student-))
```

The value of the **:conc-name** option is a symbol. It's used as the prefix to construct the names of reader functions, taking the place of the name of the entity type and the following hyphen. In our example, the reader function for the **thesis-advisor** attribute of **graduate-student** is **student-thesis-advisor**.

2.3.14. Order of Defining Pieces of a Schema

The order of the **statice:define-schema** and **statice:define-entity-type** forms doesn't make any difference. You can write them in any order you like. You don't have to do anything special about "forward references". (An example of a "forward reference" is when the **student** entity type refers to the **department** entity type before **department** is defined).

At the time **statice:make-database** is called, though, everything must be complete and consistent.

Since each **statice:define-entity-type** form defines a flavor, it is a good idea to provide a **compile-flavor-methods** form in the file, after all methods for the fla-

vor. It's even advisable if you don't define any methods of your own, because Statice itself defines methods. See the macro **compile-flavor-methods** in *Symbolics Common Lisp Dictionary*.

2.4. Coping with Transaction Restarts

This section shows how the possibility of transaction restarts can affect a program, mentions some pitfalls to avoid and some techniques to use.

One Way to Show the Students

The function **show-students-1** prints the names of all the students:

```
(defun show-students-1 ()
  (with-database (db *university-pathname*)
    (let ((name-list
           (with-transaction ()
             (let ((names nil))
               (for-each ((s student))
                 (push (person-name s) names))
               names))))
      (dolist (name name-list)
        (format t "~%~A" name)))))
```

At first, this seems like an unnecessarily awkward way to print the names. **show-students-1** enters a transaction, builds up a list of names using **statice:for-each**, and then returns the list while finishing the transaction. Then it uses **dolist** to print each of the names. Why not just print the names from inside the body of the **statice:for-each** form?

Transaction Restarts

The problem is that Statice sometimes stops midway through a transaction and starts it all over. While a transaction is in progress, any call to a Statice facility (including finishing the transaction) can make Statice decide to *restart* the transaction. When a transaction is restarted, all its side-effects on the database are undone, and control is thrown back to the beginning of the **statice:with-transaction** form (unbinding any dynamic variables and running any unwind-protect handlers on the way).

The reasons that Statice restarts transactions have to do with the underlying concurrency control and recovery system. For example, Statice will restart a transaction when two or more transactions become involved in a *deadlock* (also called a "deadly embrace") in which they're all waiting for each other. Restarts can also arise from Statice's "optimistic locking" scheme. We'll discuss later the specific reasons for restarts; it's not important for you to understand the details now.

You do have to understand that restarts can happen, and you have to be careful to write your programs to work right no matter when and where restarts occur. The most important thing to be careful about is operations inside a transaction that

have *side-effects*, other than side-effects on the database itself. If the transaction is restarted, some of those side-effects might happen more than once.

In our example, if we tried to write a version of **show-students-1** that simply printed from inside the body of the **statice:for-each** form, consider what would happen if there were a transaction restart halfway through. Some of the students would be printed twice (or more times, if there were more than one restart). In general, we mustn't do output from within a transaction.

Instead, **show-students-1** first builds up a list of the names of the students. If there is a transaction restart during the transaction, the body of the **statice:with-transaction** form starts over again, and the the variable **names** starts again from **nil**.

Another Way to Show the Students

The function **show-students-2** also prints the names of all the students. It shows another way to cope with transaction restarts.

```
(defun show-students-2 ()
  (with-database (db *university-pathname*)
    (let ((name-list nil))
      (with-transaction ()
        (setq name-list nil)
        (for-each ((s student))
          (push (person-name s) name-list)))
      (dolist (name name-list)
        (format t "~%~A" name)))))
```

The peculiar thing about **show-students-2** is the form **(setq name-list nil)**. The **setq** seems superfluous, since **name-list** is initially bound to **nil** by the **let**.

The **setq** is needed because the transaction might restart. If the **setq** were not there, and the transaction restarted, some of the same names would be pushed onto the list more than once.

2.4.1. Taking Snapshots with the :cached Attribute Option

Since many Statice programs need to deal with transaction restarts, Statice supports the technique of "taking snapshots" of portions of a database.

In the section "Coping with Transaction Restarts", we showed two ways to print the names of the students. The function **show-students-3** below shows how to do the same thing using the snapshot technique.

```
(defun show-students-3 ()
  (with-database (db *university-pathname*)
    (let ((students nil))
      (with-transaction ()
        (setq students nil)
        (for-each ((s student))
          (person-name s)
          (push s students)))
      (dolist (s students)
        (format t "~%~A" (person-name s))))))
```

show-students-3 does two unusual things. First, it calls the reader function **person-name** from within the transaction, but ignores the returned value. Second, it calls **person-name** from outside the transaction. What's going on here?

The :cached Attribute Option

The **name** attribute of **person** is defined using the **:cached** attribute option.

```
(define-entity-type person ()
  ((name string :unique t :no-nulls t :cached t
        :inverse person-named :inverse-index t)
   (id-number integer :unique t :read-only t)))
```

The **:cached** option allocates a slot inside the flavor of the entity type. (This slot, or instance variable, is internal, and cannot be used by methods.) In our example, a slot is allocated in the **person** flavor, and we call this slot the *cache slot* for the **name** attribute. Of course, the cache slot is inherited by all the descendant flavors, including the **student** flavor.

When the reader function for a **:cached** attribute is used *within* a transaction, it performs its normal reading function, *and* it puts the value of the attribute into the cache slot. When the reader function for a cached attribute is used from *outside* a transaction, it returns the contents of the cache slot.

In **show-students-3**, the first iteration (the **statice:for-each**) calls the **person-name** reader function in order to read the names from the database and store them into the cache slots. Notice that **show-students-3** calls **person-name** and ignores the returned value: the purpose of calling **person-name** is to read the values from the database and store them into the cache slots. We call this *loading* the cache slots. The second iteration (the **dolist**) calls **person-name** to get the values from the cache slots and print the values.

Using Cached Attributes

Reader functions act differently, depending on whether they are called within a transaction, or outside of one. Within a transaction, a reader gets the value of an attribute in the database. If **:cached t** is specified, the reader then loads the cache slot from the database value.

When used outside of a transaction, a reader function must be from a cached attribute. Outside a transaction, a reader does not access the database—it accesses the cache slot in the entity handle.

When an entity handle is first created, the cache slots for its cached attributes are empty (unbound). If the reader is called from outside a transaction and the slot for the cached attribute is still empty, Statice starts a little transaction that loads the cache slot, and then returns the value.

Writer functions cannot be used outside of a transaction.

Snapshots

The attribute values found in cache slots at a given time are not necessarily the same as the value in the database at that time, because some other process could have changed the value since our cache slot was loaded. We say that the cached values reflect a *snapshot* of the database state—a frozen copy of what was in the database at a certain time in the past.

It's important to remember that a cache slot holds a snapshot, because the value in the cache slot might be obsolete. As soon as you finish a transaction, some other process can access and change the values your transaction dealt with. Cache slots are a copy, not the real thing.

In practice, many attributes are read-only (whether or not they actually use the **:read-only** attribute). If the value of an attribute never changes, you know that the cached copy is still up-to-date, and you never need to read the value from the database again. In our example, **person-name** is not read-only, because we want to be able to handle changes of name (when people marry, for example). The **name** attribute rarely changes, but if we were to assume that it never changes, and never bother to reload the cache slot, we'd miss real updates in the database.

Often it is important to take a snapshot of the database and make sure that the information in the snapshot is internally consistent. This is done by loading the cache slots for many attributes within a single transaction.

2.4.2. Testing Statice Programs with Transaction Restarts

Statice programs must not have any side effects inside transactions, because transactions can spontaneously abort and restart at any time. Unfortunately, it can be rather difficult to know whether you are properly obeying this restriction. Since aborting and restarting is *possible* but *rare*, if your code does have a side effect inside a transaction, testing might not reveal the mistake.

Statice offers a way to help you test your code for robustness in the face of transaction aborting and restarting. You can set the variable **statice:*restart-testing*** to a value that triggers many aborts in many places. This feature is not guaranteed to reveal all problems, but it makes it more likely that, during testing, you will encounter them.

See the variable **statice:*restart-testing***, page 181.

2.5. Querying a Statice Database with statice:for-each

This section shows some ways you can use **statice:for-each** for associative lookup and sorting.

2.5.1. Using the :where Clause of statice:for-each

Using :where with a Simple Condition

The function **show-instructors-paid-more-than** prints the names of all instructors whose salary is greater than a specified amount.

```
(defun show-instructors-paid-more-than (this-much)
  (with-database (db *university-pathname*)
    (let ((instructors nil))
      (with-transaction ()
        (setq instructors nil)
        (for-each ((i instructor)
                   (:where (> (instructor-salary i) this-much)))
          (push (person-name i) instructors)))
      (format-textual-list instructors #'princ))))
```

show-instructors-paid-more-than resembles **show-students-2** in overall structure, building a list inside a transaction, and then printing it. The new element is the **:where** clause in the **statice:for-each** special form.

The **:where** clause makes **statice:for-each** selective. The body of **statice:for-each** is executed only for those instructors that satisfy the condition of the **:where** clause. In this example, the precondition is:

```
(> (instructor-salary i) this-much)
```

This precondition is true when the instructor's **salary** value is greater than **this-much**. The body is only executed for those instructors whose salary is greater than the given amount, and only those are put on the list to be printed.

Using :where with More than One Condition

The function **show-full-instructors-after** prints the names of all instructors whose rank is "full" and whose name alphabetically follows a given string.

```
(defun show-full-instructors-after (string)
  (with-database (db *university-pathname*)
    (let ((instructors nil))
      (with-transaction ()
        (setq instructors nil)
        (for-each ((i instructor)
                   (:where (and (equal "Full" (instructor-rank i))
                                (string-greaterp (person-name i) string))))
          (push (person-name i) instructors)))
      (format-textual-list instructors #'princ))))
```

The condition in the **:where** clause is:

```
(:where (and (equal "Full" (instructor-rank i))
             (string-greaterp (person-name i) string)))
```

A **:where** clause can have one condition, or it can have one or more conditions within a list starting with **and**. When **and** is used, all of the conditions must be true for the body to be executed.

Using :where with a Set-valued Attribute

The function **show-students-taking-course** takes a course (that is, an entity handle for a course entity) and prints the names of all students taking that course.

```
(defun show-students-taking-course (course)
  (with-database (db *university-pathname*)
    (let ((names nil))
      (with-transaction ()
        (setq names nil)
        (for-each ((s student) (:where (eq course (:any (student-courses s)))))
          (push (person-name s) names)))
      (format-textual-list names  #'princ))))
```

show-students-taking-course differs from our earlier examples because the attribute tested in the **:where** clause, namely **student-courses**, is set-valued. The **:where** clause is:

```
(:where (eq course (:any (student-courses s))))
```

When the attribute in a **:where** clause is set-valued, the conditional function is applied to each element of the set, and the condition is considered true if the function's value is true for *any* of the members of the set. The set is placed in a list with the symbol **:any** to signify that the test applies to *any* member of the set.

In this example, **student-courses** is set-valued. **:any** is used around the list that has the syntax of a call to the set-valued reader function. The condition tests to see if any course in the student's **courses** attribute is **eq** to the value of **course**.

How :any Works with and

A **:where** clause might have two comparisons, one of a single-valued attribute and one of a set-valued attribute. If we wanted to modify **show-students-taking-course** to print only those students in a given department, the **:where** clause would be:

```
(:where (and (eq course (:any (student-courses s)))
             (eq dept (student-dept s))))
```

courses is set-valued and **dept** is single-valued. Note that the **:any** only applies to the condition that tests the set-valued attribute, not the condition that test the single-valued attribute.

For more information on the **:where** clause: See the section "General Rules of the **:where** Clause of **statice:for-each**", page 43.

2.5.2. General Rules of the :where Clause of statice:for-each

Every condition is a list of three elements, according to the following template:

(*comparison form* (*reader-function variable*))

comparison	A symbol specifying the kind of comparison. These symbols are usually the names of Lisp functions, such as **>**, **equal**, or **string-greaterp**.
form	A Lisp form. It is evaluated once, before the iteration begins.
reader-function	The name of a reader function, for one of the attributes of the type being iterated over. This cannot be an inverse reader function.
variable	A symbol, the same variable used as the variable of iteration of the **statice:for-each**.

The overall syntax of the condition is designed to look like a function call, where *comparison* is usually the name of a Lisp function, and the second and third elements play the role of arguments. The syntax is intended to be natural, and easy to remember. Keep in mind that it's not really a function call, and must be in the syntax described here.

If the reader function in the third element is set-valued, the reader function must be surrounded by **:any**, so that the whole condition is of the form:

(*comparison form* (:any (*reader-function variable*)))

Interchanging the "Arguments"

The second and third elements of the list can be interchanged, for some kinds of comparisons. For example, in **show-full-instructors-after**, the **equal** condition was in the usual order, with the Lisp form first, but the **string-greaterp** condition was in the reverse order, with the two-element sublist first. In such a case, the condition is of the form:

(*comparison* (*reader-function variable*) *form*)

The ordering of the second and third elements doesn't matter for commutative comparison functions such as **equal**, but it does matter for non-commutative ones such as **string-greaterp**. For example, the following two conditions mean the same thing:

```
(string-greaterp (person-name i) string)
(string-not-greaterp string (person-name i))
```

string-search is a kind of comparison for which the order must not be reversed. A typical example condition using **string-search** is:

```
(string-search string (person-name i))
```

This condition selects only those people whose name contain **string** as a substring.

Exception for typep

There is one exception to the above syntax, for a special kind of comparison called **typep**. In a **typep** condition, the second element is simply the name of the variable of the **statice:for-each**, and the third element is a form that evaluates to an entity type name. Example:

```
(for-each ((s student)
           (:where (and (typep s 'graduate-student) ...)))
  ...)
```

This condition selects only those students that are of type **graduate-student**.

2.5.3. Using the :count Clause of statice:for-each

Sometimes, you want the body of **statice:for-each** to be executed for only one entity. The **:count** option of **statice:for-each** allows the caller to limit the number entities for which the body is called. By using the **:count** option, you can reduce consing and improve query performance. For example:

```
(for-each ((p person))
  (return p))
```

could be improved by:

```
(for-each ((p person) (:count 1))
  (return p))
```

2.5.4. Sorting Entities with the :order-by Clause of statice:for-each

The function **show-courses-in-dept-sorted** prints the names of all courses in the given department, in alphabetical order by title.

```
(defun show-courses-in-dept-sorted (dept)
  (with-database (db *university-pathname*)
    (let ((titles nil))
      (with-transaction ()
        (setq titles nil)
        (for-each ((c course)
                   (:where (eq (course-dept c) dept))
                   (:order-by (course-title c) descending))
          (push (course-title c) titles)))
      (format-textual-list titles #'princ))))
```

What's new here is the **:order-by** clause:

```
(:order-by (course-title c) descending)
```

The **:order-by** clause does not affect the set of entities that **statice:for-each** iterates over, but it controls the *order* of the iteration. The entities are sorted by the values of an attribute, and the iteration is done in that order.

In this example, **statice:for-each** iterates over **course** entities. The **:order-by** clause tells **statice:for-each** to iterate in alphabetical order according to the value

of the **title** attribute of **course**. The order is specified as **descending** because **push** builds lists in reverse order, so that once the list is finished, it's really in ascending order. The final list, **titles**, is in ascending alphabetical order.

2.5.5. Using statice:for-each on Many Variables

The function **instructors-in-dept-headed-by** returns the names of all instructors who are in a department headed by a particular person.

```
(defun instructors-in-dept-headed-by (head)
  (with-database (db *university-pathname*)
    (let ((instructors nil))
      (with-transaction ()
        (setq instructors nil)
        (for-each ((i instructor) (d department)
                   (:where (and (eq d (instructor-dept i))
                                (eq (department-head d)
                                    (person-named head)))))
          (push (person-name i) instructors)))
      instructors)))
```

The **statice:for-each** form has two variables, **i** and **d**. **statice:for-each** iterates over every possible pairwise combination of values for **i** and **d**. In other words, **statice:for-each** considers each instructor, and for each instructor it considers each department. In mathematical jargon, it takes the Cartesian product of instructors and departments.

When it has a pair of **i** and **d**, it examines the **:where** clause to decide whether to run the body. The **:where** clause can refer to both variables, **i** and **d**. The second condition in the **:where** clause is the familiar kind that tests the value of an attribute. The first condition is a new kind that joins the two variables. In English, the entire **:where** clause means "If **d** is the department of **i**, and the head of **d** is the person named **head**". We call this a *join condition*.

Within the body of the **statice:for-each** both variables **i** and **d** are bound to entity handles, and the body can use either or both. In our example, the body doesn't need to use **d** for anything; **d** played its useful part inside the **:where** clause.

2.6. Using Indexes to Increase Database Performance

2.6.1. Introduction to Indexes in Statice

In this section we introduce a new concept: the *index*. Indexes exist inside Statice databases, but they don't represent any new information. In fact, any program will return the same results regardless of what indexes exist. Even though they have no effect on the results, indexes are important because they have a great effect on performance.

It's important to use indexes. If you don't, your program will become slower and slower as the size of your database grows. Fortunately, indexes are easy to use:

you just create them, and ignore them. And any time you change your mind, it's easy to create or destroy them. In fact, you can create an index, try running a program, then destroy the index and try again, and observe the effects on performance.

Indexes have drawbacks as well as advantages. In general, they make lookups and searches much faster, but they make inserting and deleting slightly slower.

Indexes and Inverse Reader Functions

Consider the inverse reader **person-named**. How does it find the entity for the person named "Joe Cool"? It could examine each entity of type **person** (including entities whose type inherits from type **person**), check the value of the **name** attribute, and see whether it equals "Joe Cool". This would be slower and slower as the database grew, taking time linearly proportional to the number of **person** entities. In fact, this is what happens if there isn't an index.

To make **person-named** fast, we can make an index. An index is auxiliary structure that resides invisibly in the database. This particular index is a compact table that represents a mapping from values of the **name** attribute to the corresponding entities, sorted alphabetically and generally organized to make searching fast. (In database jargon, it's organized as a *B+ tree*.) If this index exists, **person-named** will automatically use the index to find the entity handle.

To see how the index was created, look at the description of the **name** attribute of the **person** entity type.

```
(define-entity-type person ()
  ((name string :unique t :no-nulls t :cached t
         :inverse person-named :inverse-index t)
   (id-number integer :unique t :read-only t)))
```

The **:inverse-index** option means that when the database is created, an index (initially empty) should also be created. This kind of index is specified by **:inverse-index** because it speeds up the inverse reader function.

Indexes and Accessor Functions

Here's an example of an index used to speed up a regular accessor function, rather than an inverse reader function. Notice that the description of the **courses** attribute of the **student** entity type includes the **:index t** option:

```
(define-entity-type student (person)
  ((dept department :inverse students-in-dept)
   (courses (set-of course) :index t :inverse course-students)
   (shirts (set-of shirt) :unique t :inverse shirt-owner)))
```

The index specified for the **courses** attribute speeds up the performance of both the reader **student-courses** and the corresponding writer (which is called with **setf** of **student-courses**). The performance improvement happens whether you are accessing the attribute for reading or writing, because it increases the speed of locating the attribute in the database.

How does the **student-courses** reader function work? If there is no index, Statice must do a search through a sequence of small data structures called *tuple records*, of which there is one for every relationship between a student and a course. Again, this takes time linear in the number of such relationships, and so **student-courses** gets increasingly slower as the database grows.

The **:index** option causes Statice to make an index that makes **student-courses** fast. This is a different kind of index internally; it gives each **student** entity a list of pointers to the **course** entities. But all these details are invisible to you. You just make the index, and **student-courses** automatically runs faster. Note that you could also use the **:inverse-index** option this attribute to speed up **course-students**.

The **:index** option is only meaningful for set-valued attributes. For a single-valued attribute, there's no need for an index, since Statice can obtain the value directly without searching. (There's one case in which it does make sense to use **:index** with a single-valued attribute, involving areas: See the section "Statice Type Sets, Attribute Sets, and Areas", page 127.)

Indexes are also useful for speeding up the associative queries done with **statice:for-each**. For more information: See the section "Indexes and **statice:for-each**", page 47.

We present indexes in more detail later: See the section "Statice Indexes", page 120.

2.6.2. Indexes and statice:for-each

How does **statice:for-each** find the entities that conform to the conditions of the **:where** clause? Look again at the function **show-instructors-paid-more-than**, which prints the names of all instructors whose salary is greater than a specified amount.

```
(defun show-instructors-paid-more-than (this-much)
  (with-database (db *university-pathname*)
    (let ((instructors nil))
      (with-transaction ()
        (setq instructors nil)
        (for-each ((i instructor)
                   (:where (> (instructor-salary i) this-much)))
          (push (person-name i) instructors)))
      (format-textual-list instructors #'princ))))
```

Without an index, **statice:for-each** actually examines each instructor entity, and looks at the value of the salary attribute to see whether it's greater than **this-much**. Does this make **statice:for-each** much slower? It depends. If **this-much** is a very small number, **statice:for-each** has to deal with almost all the instructors anyway, so the search does not slow things down significantly. But if **this-much** is a very large number, **statice:for-each** only really needs to deal with a few instructors, and the extra overhead of examing and rejecting the rest could slow down the query significantly.

To make **statice:for-each** faster, we can add an inverse index to the **salary** attribute of **instructor**. **statice:for-each** automatically notices that this index exists. The index is internally stored in sorted order, so **statice:for-each** can efficiently find pointers to all the instructors whose salary is greater than **this-much**. (In database jargon, this **statice:for-each** is doing a *range query*, searching for values that fall within a given range. The range, in this example, is from **this-much** to infinity. One reason Statice uses the *B+ tree* organization is because it works well for range queries.)

Why is the **statice:for-each** sped up by an inverse index, rather than a regular index? Because the job of an inverse index is to get from the value to the entity, while the job of a regular index is to get from the entity to the value. The **statice:for-each** has the value (actually a range of values), and it's trying to find the entities. So an inverse index is the right kind.

Here's another way to explain why we're using an inverse index instead of a regular index. Suppose the **salary** attribute of **instructor** had an inverse reader function, named **instructors-whose-salary-is**. It would be a function of one argument, a salary amount (a floating point number), that returns a list of instructors. Another way to do the same thing would be the write a function using **statice:for-each**:

```
(defun instructors-whose-salary-is (salary-value)
  (let ((result nil))
    (for-each ((i instructor)
               (:where (= (instructor-salary i) salary-value)))
      (push i result))
    result))
```

Such a function does the same thing that an inverse reader function would do. As you can see, it's similar to **show-instructors-paid-more-than**. **statice:for-each** is given a value, and it's supposed to find the entity. When we're going in that direction, we need an inverse index. By the way, you can have an inverse index on an attribute even if there is no any inverse reader function.

2.6.3. statice:for-each Can Use Many Indexes Together

The function **find-big-blue-shirt** returns an entity handle for a shirt whose size is greater than 15 and whose color is "blue", or returns **nil** if there is no such shirt.

```
(defun find-big-blue-shirt ()
  (with-database (db *university-pathname*)
    (with-transaction ()
      (for-each ((s shirt)
                 (:where (and (> (shirt-size s) 15)
                              (string-equal "blue" (shirt-color s)))))
        (return s)))))
```

How does **statice:for-each** go about evaluting this query? It depends on what indexes exist. If, as in our example schema, there are no useful indexes, it simply checks each shirt to see whether the shirt meets both criteria.

If we make an inverse index on the **size** attribute of **shirt**, **statice:for-each** uses that index to find all the big shirts, and then examine each of them to see whether it's blue. This is much faster, in a large database. Similarly, if there is an index on the **color** attribute, **statice:for-each** uses that index to find all the blue shirts, and then examine each of them to see whether it's big.

If we make an index on both the **shirt** attribute and the **color** attribute, **statice:for-each** uses *both* indexes. Each index yields a set of entities, and **statice:for-each** computes the intersection of the sets. This is the fastest way of all, in a large database.

In general, if a **statice:for-each** has a **:where** clause with several conditions **and**'ed together, **statice:for-each** examines each condition to see whether there's an index that can resolve that condition. It uses all those indexes, and computes the intersection of all the resulting sets. Then, if there are any conditions left over that didn't have an index, **statice:for-each** checks each entity to make sure that it meets all of the leftover conditions as well.

2.6.4. Making and Deleting Indexes

You can make new indexes or delete existing indexes at any point within a transaction, even while other processes are using the database at the same time. The concurrency control mechanism of Statice ensures that none of the results of the transactions conflict with one another. If a large number of entities of the type already exist, it might take some time for Statice to make the index.

Statice provides functions for making and deleting indexes, and testing whether indexes exist. These functions expect the name of the reader function as their argument; this identifies the index.

The function **statice:make-index** makes an index. For example, the following form makes an index on the **shirts** attribute of **student**:

```
(make-index 'student-shirts)
```

The function **statice:delete-index** deletes an index. For example, this form deletes the index on the **courses** attribute of **student**:

```
(delete-index 'student-courses)
```

The function **statice:index-exists** returns **t** if an index exists, and **nil** if it does not. For example, the following form asks whether there is an index on the **courses** attribute of **student**:

```
(index-exists 'student-courses) => nil
```

Statice provides analogous functions for inverse indexes. **statice:make-inverse-index**, **statice:delete-inverse-index**, and **statice:inverse-index-exists**. These all take as their argument the name of the reader function. Note that they take the name of the regular reader function, *not* the name of the inverse reader function.

For example:

```
(make-inverse-index 'shirt-size)
(delete-inverse-index 'person-name)
(inverse-index-exists 'course-instructor) => nil
```

2.6.5. Indexes and :order-by

We've seen how indexes can speed up the **:where** clause of **statice:for-each**. They can also speed up the **:order-by** clause. Here's an example. The function **show-names-sorted** prints the names of all people in alphabetical order.

```
(defun show-names-sorted ()
  (with-database (db *university-pathname*)
    (let ((names nil))
      (with-transaction ()
        (setq names nil)
        (for-each ((p person)
                   (:order-by (person-name p) descending))
          (push (person-name p) names)))
      (format-textual-list names #'princ))))
```

If there is no inverse index on the **name** attribute, **statice:for-each** has to find all the entities and then sort them. However, there *is* such an index. As we said earlier, the index entries are sorted by the value of the **name** attribute. Since the entries are already in properly sorted order, **statice:for-each** doesn't have to do a time-consuming sort operation. It uses the index, to find all the entities already sorted into order.

If a **statice:for-each** has both a **:where** clause and an **:order-by** clause, and both can be helped by indexes, **statice:for-each** uses both indexes. Consider the function **show-new-names-sorted**, which prints out the names of all people whose ID number is greater than 1000, sorted alphabetically:

```
(defun show-new-names-sorted ()
  (with-database (db *university-pathname*)
    (let ((names nil))
      (with-transaction ()
        (setq names nil)
        (for-each ((p person)
                   (:where (> (person-id-number p) 1000))
                   (:order-by (person-name p) descending))
          (push (person-name p) names)))
      (format-textual-list names #'princ))))
```

If there is an inverse index on **id-number** and an inverse index on **name**, **statice:for-each** uses both indexes together, one to find the subset of entities to process, and the other to control the order in which to process them.

2.7. Multiple Indexes

2.7.1. Introduction to Multiple Indexes

A *multiple index* is an index on two or more attributes. The **course** entity type has a multiple index on attributes **title** and **dept**.

```
(define-entity-type course ()
  ((title string :inverse courses-entitled)
   (dept department)
   (instructor instructor))
  (:multiple-index (title dept) :unique t))
```

This multiple index is a compact table (a B+ tree) that associates *pairs* of attribute values with pointers to entities. The index entries are sorted by the values of **title**, and groups of entries that all have the same value of **title** are sorted within the group by **dept**.

There are two restrictions on multiple indexes:

1. The attributes must all be single-valued, not set-valued.

2. The attributes must all be from the entity type itself, not inherited from parent entity types.

Basic Use of Multiple Indexes

Here's an example of how **statice:for-each** can use a multiple index. The function **find-course** takes a title string and a department entity, and returns the designated course, or **nil** if there isn't any.

```
(defun find-course (title dept)
  (with-database (db *university-pathname*)
    (with-transaction ()
      (for-each ((c course)
                 (:where (and (string-equal (course-title c) title)
                              (eq (course-dept c) dept))))
        (return c)))))
```

When **statice:for-each** examines this query, it recognizes that the multiple index can resolve the whole query, *both* conditions, in a single step. It looks up both the title and the department in the multiple index, and finds an answer if any exists.

If there had been inverse indexes on both **title** and **course**, and no multiple index, **statice:for-each** could have used both indexes and intersected the results. However, using the multiple index is faster. In fact, if the multiple index had existed *and* both of the inverse indexes had existed, **statice:for-each** would have chosen to use the multiple index. In general, **statice:for-each** looks over the different ways of resolving a query, and automatically chooses the one it anticipates will be fastest.

Uniqueness Constraints

In addition to speeding up queries, multiple indexes can impose uniqueness constraints on entity types. The **:unique t** in the **:multiple-index** option means that this index imposes a uniqueness constraint. The constraint states that no two distinct courses can have *both* the same title *and* the same department. In other words, the constraint means that given a particular title and a particular department, there can be at most one course with that title and that department. (For those familiar with the terminology of relational data models, this can be compared with the notion of "composite candidate keys".)

When null values are involved (See the section "The Statice Null Value", page 29.) we follow the same principle as for unique attributes: for purposes of uniqueness constraints, one null value *does not equal* another null value. For example, there could be two distinct courses in the same department, both of whose titles are null. The idea is that they don't have titles yet, so they aren't really conflicting with each other.

Earlier we said that indexes have no semantic effect, and that they only affect performance. The uniqueness feature described here is the sole exception to that rule.

Making and Deleting Multiple Indexes

Statice provides functions for making and deleting multiple indexes, and testing whether a multiple index exists. These functions take one required argument, a list of the names of the reader functions for the attributes.

The function **statice:make-multiple-index** makes a multiple index. For example, the following form makes a multiple index on the **size** and **color** attributes of the **shirt** entity type:

```
(make-multiple-index '(shirt-size shirt-color))
```

To make the index impose a uniqueness constraint, use the **:unique** keyword argument:

```
(make-multiple-index '(shirt-size shirt-color) :unique t)
```

The function **statice:delete-multiple-index** deletes a multiple index. For example, to delete the multiple index on **title** and **dept**:

```
(delete-multiple-index '(course-title course-dept))
```

There is no **:unique** keyword argument; **statice:delete-multiple-index** just deletes the index, whether or not it is unique.

The function **statice:multiple-index-exists** returns **t** if an index exists, and **nil** if it does not. For example, the following form asks whether there is a multiple index on the **title** and **dept** attributes of **course**:

```
(multiple-index-exists '(course-title course-dept))
```

Several different kinds of queries can take advantage of multiple indexes:

See the section "Multiple Indexes and Leading Subsequences", page 53.
See the section "Multiple Indexes and Suffix Comparisons", page 54.

See the section "Multiple Indexes and **:order-by**", page 55.

2.7.2. Multiple Indexes and Leading Subsequences

Consider the function **find-courses-with-title**, which returns a list of all courses with a given title:

```
(defun find-courses-with-title (title)
  (with-database (db *university-pathname*)
    (with-transaction ()
      (let ((result nil))
        (for-each ((c course)
                   (:where (string-equal (course-title c) title)))
          (push c result))
        result))))
```

find-courses-with-title is like **find-course**, but it discriminates only on the basis of the **title** attribute, rather than on both **title** and **dept**. However, **statice:for-each** can still use the multiple index. In the multiple index, the entries of the index are sorted first by the value of the **title** attribute. So all the entries for a particular title are grouped together, and all these groups are sorted by title. **statice:for-each** simply finds the whole group of entries for given title string, which is exactly the information being sought.

By the way, **find-courses-with-title** is essentially the same thing as an inverse reader function for the **title** attribute of **course**. If we make an actual inverse reader function, it uses the multiple index in the same way that **find-courses-with-title** does. If there is an inverse index on the **title** attribute, the inverse function uses it in preference to the multiple index, but so does **find-courses-with-title**.

This trick would not work for a query that discriminated on **dept** but not **title**, because the index entries are grouped together based on **title**. In general, if there is a multiple index on a sequence of attributes, **statice:for-each** first looks for an equality condition for the first attribute. If one is found, it then looks for an equality condition on the second attribute, and so on. Thus, **statice:for-each** then resolves all of these conditions with one lookup in the multiple index. Any remaining conditions cannot be resolved with the multiple index. In other words, **statice:for-each** looks for a set of equality conditions whose attributes form a leading subsequence of the sequence of attributes in the index.

Here's an artificial example:

```
(define-entity-type example ()
  ((a string)
   (b string)
   (c string)
   (d string)
   (e string))
  (:multiple-index (b e a c)))
```

```
(for-each ((ex example)
           (:where (and (equal (example-e ex) "huey")
                        (equal (example-c ex) "dewey")
                        (equal (example-b ex) "louie"))))
  ...)
```

statice:for-each first looks for an equality condition on the first attribute of the index, namely **b**. It finds one, namely **(equal (example-b ex) "louie")**. Next, it looks for an equality condition on **e**, and finds one. Next, it looks for an equality condition on **a**; there isn't any, so it stops looking. **statice:for-each** uses the multiple index to resolve the equality conditions on **b** and **e**, and gets back the set of all entities meeting those two conditions. Then it tests each entity to see if it meets the condition on **c**. Even though **c** is one of the attributes of the multiple index, **statice:for-each** can't use it, because it's not part of a leading subsequence.

2.7.3. Multiple Indexes and Suffix Comparisons

The function **show-well-paid-english-profs** prints the names of all instructors in the English department who earn more than 50000.

```
(defun show-well-paid-english-profs ()
  (with-database (db *university-pathname*)
    (let ((names nil))
      (with-transaction ()
        (setq names nil)
        (for-each ((i instructor)
                   (:where (and (eq (instructor-dept i)
                                    (department-named "English"))
                                (> (instructor-salary i) 50000))))
          (push (person-name i) names)))
      (format-textual-list names #'princ))))
```

Suppose there is a multiple index on attributes **dept** and **salary** of the **instructor** entity type. The **statice:for-each** does an equality comparison on the value of the **dept** attribute, and a greater-than comparison on the **salary** attribute.

statice:for-each can resolve the query in one step, using the multiple index. Remember how the index entries are arranged. First, they're sorted by the value of **dept**. Then, within each group of entries for which **dept** is equal, they are sorted by the value of **salary**. This means that all the index entries for the entities we're looking for are stored together, contiguously, in the index. **statice:for-each** can simply find that range of the index, and it has its answer.

This would not have worked if the multiple index were on **salary** and **dept**. It's important that **dept** was the first attribute in the index, because the index entries are first sorted by **dept**.

In general, **statice:for-each** can do this kind of query resolution using a multiple index if, and only if, there is an equality condition in the **for-each** for all the attributes of the multiple index except the last one, and there is a comparison (or **string-prefix**) condition in the **for-each** for the last attribute of the multiple index.

Multiple indexes can also help with sorting: See the section "Multiple Indexes and **:order-by**", page 55.

2.7.4. Multiple Indexes and :order-by

Earlier we saw that indexes can speed up the **:order-by** clause of **statice:for-each**. (See the section "Indexes and **:order-by**", page 50.) Multiple indexes can do this, too. The function **get-sorted-shirts** returns a list of shirt entities, sorted first alphabetically by the name of the color, and then (within shirts of a given color) by size.

```
(defun get-sorted-shirts ()
  (with-database (db *university-pathname*)
    (let ((shirts nil))
      (with-transaction ()
        (setq shirts nil)
        (for-each ((s shirt)
                   (:order-by (shirt-color s) ascending
                              (shirt-size s) ascending))
          (push s shirts)))
      (nreverse shirts))))
```

If there is a multiple index on the **color** attribute and the **size** attribute, **statice:for-each** uses the index directly to get its answer. You can picture one such index as follows:

blue, size 1
blue, size 2
blue, size 5
red, size 1
red, size 2
red, size 10

This index is the ideal index for **get-sorted-shirts**, because the index entries are in exactly the order that **get-sorted-shirts** is interested in.

Even if we didn't have the ideal index, we might have an index that is helpful. If there is a multiple index on the **color** attribute and the **washed** attribute, **statice:for-each** uses the index to accomplish the first part of the sorting. The index provides **statice:for-each** with groups of shirt entities, with a particular color for each group, and these groups are sorted by color. Within each group, **statice:for-each** must sort the entities by size. Although this is not as good as the ideal index, it is still a lot faster than sorting the entire set of entities by both color and size.

In general, **statice:for-each** can use a multiple index in this way if the first one-or-more of its attributes appear in the **:order-by** clause (and they're either all ascending or all descending).

There's another interesting case where **statice:for-each** takes advantage of a multiple index, this time to deal with a **:where** clause and an **:order-by** clause both at

the same time. Consider the function **show-courses-in-dept-sorted**, which we introduced earlier (See the section "Sorting Entities with the **:order-by** Clause of **statice:for-each**", page 44.).

```
(defun show-courses-in-dept-sorted (dept)
  (with-database (db *university-pathname*)
    (let ((titles nil))
      (with-transaction ()
        (setq titles nil)
        (for-each ((c course)
                   (:where (eq (course-dept c) dept))
                   (:order-by (course-title c) descending))
          (push (course-title c) titles)))
      (format-textual-list titles #'princ))))
```

Suppose there is a multiple index on the **dept** attribute and the **title** attribute. The **statice:for-each** has an equality condition on the **dept** attribute, and needs to sort using the **title** attribute. It turns out that the multiple index has exactly what the **statice:for-each** needs. The index entries in the multiple index are grouped, with each entry's entity having the same value of **dept**. Within that group, the entries are sorted by **title**. This is exactly what the **statice:for-each** needs, so it gets it straight from the index.

The general rule for this kind of query resolution is that a multiple index can be used if the first one-or-more attributes are used in equality conditions by the **statice:for-each**, and all the rest of the attributes are used in the **:order-by** clause (and they're either all ascending or all descending).

3. Advanced Techniques for Statice Applications

3.1. Hints and Techniques for Using Statice

3.1.1. Choosing the Forward Direction for a Statice Schema

When you are designing a schema, you can express a relationship between a pair of entity types in two ways, depending on which direction is forward and which is backward. Consider the relationship between students and courses in the University example:

```
(define-entity-type student (person)
  ((dept department :inverse students-in-dept)
   (courses (set-of course) :index t :inverse course-students)
   (shirts (set-of shirt) :unique t :inverse shirt-owner)))

(define-entity-type course ()
  ((title string :inverse courses-entitled)
   (dept department)
   (instructor instructor))
  (:multiple-index (title dept) :unique t))
```

(The university example is presented in full elsewhere: See the section "Defining a Schema for a University", page 20.)

The relationship between students and courses is modeled by the **courses** attribute of the **student** entity type. **student-courses** is the accessor function, and **course-students** is the inverse accessor function.

Here is an alternate way to model the same thing:

```
(define-entity-type student (person)
  ((dept department :inverse students-in-dept)
   (shirts (set-of shirt) :unique t :inverse shirt-owner)))

(define-entity-type course ()
  ((title string :inverse courses-entitled)
   (dept department)
   (instructor instructor)
   (students (set-of student) :inverse-index t :inverse student-courses))
  (:multiple-index (title dept) :unique t))
```

Here the same relationship is modeled by the **students** attribute of the **course** entity type. **course-students** is the accessor function, and **student-courses** is the inverse accessor function, which is just the opposite of what we first saw. This time we had to use **:inverse-index** rather than **:index**.

However, the functions **course-students** and **student-courses** do exactly the same thing, regardless of the choice you make about the schema. The index is still designed to speed up **student-courses**. When you use the accessor functions, it doesn't make any difference whether they are regular or inverse accessor functions.

Whenever you model a relationship between two entity types, you can put the attribute into either type. For the most part, it doesn't matter which way you choose, because you can always make inverse accessors, inverse indexes, and so on. Usually you can pick whichever one seems most expressive to you.

It is important, however, not to duplicate the same information by representating a single relationship between two entity types in two distinct attributes; such duplication is unnecessary and error-prone. In this sense, Statice is different from Lisp. In Lisp, you can follow the forward direction, but not the backward direction, but in Statice, you can go in both directions. For example, if you define **student** to have an attribute **course** with an inverse reader, you can get from a course entity the students taking the course, and there is no need to define **course** to have an entity **students**.

One circumstance in which the choice of direction matters is when you want to use a multiple index: See the section "Multiple Indexes", page 51. In the example above, there is a multiple index on attributes **title** and **dept** of **course**. If the relationship between **department** and **course** had been modeled by a **course** attribute of the **department** entity type, there would be no way to make the multiple index.

3.1.2. Representing Information as an Ordinary Value Versus an Entity

When designing a schema, you should think about which kind of information should be modeled as entities, and which simply as ordinary values, such as string or integers.

For example, suppose you had an entity type called **person**, and you wanted to store the political party in which the person is registered. So you want the entity-type **person** to have an attribute named **party**. What should be the type of the attribute? You might make it a string, and store "Republican" or "Democrat" as the value. This is simple and straightforward. However, someday you might want to store attributes of political parties, such as the year the party was founded. In that case, it would be better to have an entity type called **party**, and have some entities of that type, one representing the Republicans and one representing the Democrats and so on, so that you could store attributes of each party.

In general, you should try to think ahead and anticipate how the database might grow in the future. By using entities to model things that are distinct objects or concepts in the real world, you make it easier to expand the database schema in the future.

3.1.3. Warning About Changing the Package of a Statice Program

If you change the way your application uses Lisp packages, Statice can get confused. This section describes two sources of confusion, and the solutions to them.

Inside each database, the name of the schema is stored. For example, suppose you make a new database by doing:

```
(make-database #p"foo:>a>b" 'my-package:my-schema)
```

Inside the database **foo:>a>b**, Statice remembers that the symbol **my-package:my-**

schema is the name of the schema. When you open the database later, Statice uses this information to locate the template schema information. For background information: See the section "Template Schemas and Real Schemas", page 110.

When You Move a Statice Program From One Package to Another

You might later decide to move your program from the package called **my-package** to a different package called **other-package**. You could do this by moving code from one file to another, or by changing the file attribute lists of an existing file. However, your databases still "know" that the name of the template schema is **my-package:my-schema**. If you try to use these databases, Statice will look in the **my-package** package for the symbol **my-schema**, and won't find it. If the package **my-package** no longer exists, Statice will signal an error trying to intern a symbol in **my-package**. If the package does still exist, Statice will signal the **statice:schema-not-loaded** error. The error message is:

```
The database foo:>a>b contains the symbol my-package:my-schema,
and expects to find that my-package:my-schema is defined,
in the Lisp environment, as a Statice schema.
However, it is not.  Perhaps the application
program is not loaded, not all loaded, or loaded
into some other package than my-package.
If the schema is in fact loaded, and the symbol contained
in the database (shown above) is incorrect, you may use the
Set Database Schema Name Command to change it.
```

To fix the database to know that its schema name is now really **other-package:my-schema**, use the following command:

```
Set Database Schema Name foo:>a>b other-package:my-schema
```

Do this for every existing database that uses the schema whose package was changed.

See the section "Set Database Schema Name Command", page 170.

When You Kill and Redefine the Package of a Statice Program

Here we discuss a problem that can happen when you are writing a Statice program in a package (called **my-package**), and, for some reason or another, you kill that package (using **pkg-kill**), and then redefine the **my-package** package.

Now, if a Statice database refers to the schema **my-package:my-schema**, and we open the database, and then we kill the package **my-package**, and make a new package called **my-package**, and reload everything into that new package, Statice is still holding onto the old symbol that was interned in the old package. So Statice won't use the current **my-package:my-schema**.

The solution for this is to *terminate* the database and then open it again, which will make Statice look up the symbol again. Statice calls **intern** anew and finds the new symbol.

statice:terminate-database takes one argument, the pathname where the database is stored.

See the function **statice:terminate-database**, page 216.

3.1.4. Obtaining a Symbol From a Database, When the Package is Undefined

Statice signals an error if if you try to get a symbol such as **foo:bar** out of a Statice database, and there is no package named **foo** defined in your Lisp world.

```
Error: FOO is not meaningful as a package name.
```

This error offers several proceed options.

This can happen if two clients are using the same database: client A has a package named **foo** in its Lisp world but client B does not; client A stores a symbol from package **foo** into the database; and client B tries to read it.

Another way it can happen is with only one client: if client A defines package **foo**, stores a symbol from package **foo** into the database, then client A cold-boots, and tries to read that symbol out of the database but this time it has not defined package **foo**.

To be more specific, what happens is that Statice signals the error **statice:symbol-package-not-found**. This error is built upon the error **sys:package-not-found**.

3.1.5. Guide to the Statice Examples

Statice comes with several files of examples, which are in the directory SYS:STATICE;EXAMPLES;. This section explains each of the example files.

SYS:STATICE;EXAMPLES;BANK-EXAMPLE.LISP
: The bank example used in this document. See the section "Quick Overview of Statice: the Bank Example", page 3.

SYS:STATICE;EXAMPLES;UNIVERSITY-EXAMPLE.LISP
: The university example used in this document. See the section "A More Complicated Schema: the University Example", page 20.

SYS:STATICE;EXAMPLES;PRESENTATION-TYPE.LISP
: The definition of the **statice-utilities:entity-named-by-string-attribute** presentation type; an example of a presentation type for entities. See the section "Presentation Type for Statice Types with Simple String Names", page 106.

SYS:STATICE;EXAMPLES;BOOKS.LISP
: A sample Statice application program, using Genera's user interface management system, that does simple bookkeeping. This is also a good example for many Dynamic Windows facilities.

SYS:STATICE;EXAMPLES;FINGER-HACK.LISP

A daemon program that runs on every client at a site. The daemon periodically updates a shared Statice database, keeping track of who is logged in and what the user's idle time is. This program features a heavy degree of concurrent access to a shared database. It has been run with up to 100 clients.

SYS:STATICE;EXAMPLES;IMAGE.LISP

Some simple examples of how to use the **statice:image** type. See the statice type specifier **statice:image**, page 77.

SYS:STATICE;EXAMPLES;EXTENDED-TYPES.LISP

Several examples of extended type definitions, including those used in this document. See the section "Defining New Statice Types", page 84.

SYS:STATICE;EXAMPLES;WEATHER-REPORT-COMMAND.LISP

This program gathers and manipulates weather data. It provides various user interfaces and analysis routines, and accumulates the weather data in a Statice database. Since this program relies on the existence of a network server that provides the raw weather data, it is unlikely that you can run this program, but it utilizes many interesting techniques.

SYS:STATICE;EXAMPLES;WEATHER-JOSHUA.LISP

This is a sequel to the **weather-report-command** example. It uses Joshua to analyze the weather data, and run simple rules to predict the weather. This example shows one way to define a Joshua predicate model that accesses a Statice database, thus tying the expert system facilities of Joshua to the database facilities of Statice.

3.1.6. Checking for Disk Write Errors

Statice has an optional mode in which it checks for disk write errors. When disk write error checking is turned on, Statice follows every disk write operation with a disk read operation: it reads back the data that it just wrote, and makes sure that the data reads back without causing a disk system error (such as an ECC error).

This mode is useful because disk subsystems are sometimes unreliable. If a block is written with incorrect data, a database could be rendered useless, and have to be restored from backup tape. The disadvantage of this mode is that it makes writing slower: every disk write operation now must wait for the disk to spin around again and for a disk read operation to happen.

On hardware models 3600, 3640, 3645, 3670, and 3675, there has been a problem observed in the disk subsystem in which disk writing somehow omits one word of data. Blocks that are written this way will cause an ECC error when they are read back. The problem is very rare and very hard to reproduce, but it does happen. Hardware models 3610AE, 3620, 3650, and 3653 have a very different disk subsystem, and this problem has not been observed. Of course, no hardware is perfect.

The mode is controlled by the value of the global variable **dbfs:*check-writes***. The value is interpreted as follows:

nil	No disk write error checking is done.
t	Disk write error checking is always done.
:obs	Disk write error checking is only done if the machine is of hardware model 3600, 3640, 3645, 3670, or 3675. This is the default.
:nbs	Disk write error checking is only done if the machine is of hardware model 3610AE, 3620, 3650, or 3653.

If a disk write error is detected, a notification is produced with a message like this:

```
[01:43:22 DISK WRITE ERROR: Incorrect data written to DBFS log
    causing the following error: (write will be retried)
    %DISK-ERROR-ECC during a %DCW-READ32
             on unit 0., cyl 294., surf 2., sec 23.,
   Fatal ECC error,
   3. pending transfers associated with this disk event aborted.]
```

Note: if you get this notification, you don't have to do anything about it. It just means that Statice detected the disk write error and intends to fix it. You can safely ignore the notification.

After sending the notification, Statice retries the write operation. If it fails again, another notification is sent, and so on; if it succeeds, Statice continues normal operation. It keeps retrying until it has tried more than **dbfs:*check-writes-retry-limit*** times; the default value of this variable is 10. If the operation is still failing after that, Statice signals an error:

```
Retry limit exceeded while trying to propagate changes
from the DBFS log to the database files.
```

It offers a proceed type called "Continue trying to rewrite data" which tries the writes another **dbfs:*check-writes-retry-limit*** times. If the error persists, Statice cannot continue to operate; your hardware must be repaired.

3.2. Browsing a Statice Database

The Statice Browser is a facility for viewing and modifying entities in a database.

Selecting the Statice Browser

The Browser is started by pressing `SELECT B`, or by selecting Statice Browser from the system menu.

Panes of the Statice Browser

When the Browser starts up, there are six panes in the frame. The format of the screen resembles that of Document Examiner. The panes include:

Title	Displays the title Statice Browser.
Viewing	Entities are displayed and modified here; this is the largest pane.
Current Candidates	Shows all the entities which are in the various queries.
Queries	Shows all the queries which have been performed so far. A query is a group of entities which have been selected from the database.
Commands	Displays the commands you enter.
Command Menu	Displays the Statice Browser commands.

Choosing a Database to Browse

To start, you should use the Open Database command to select a database to browse.

Before using the Browser, the Lisp world must have the package of the schema defined, and the schema itself defined. If the package for the schema as indicated in the database is not defined in the Lisp world, then the Browser will signal the following error:

```
Error: DEMO is not meaningful as a package name.
```

If the package is present, but the schema is not loaded, then the Browser signals the following error:

```
Error: The database RUT:>lamb>university contains the
       symbol DEMO:UNIVERSITY, and expects to find that
       DEMO:UNIVERSITY is defined, in the Lisp environment,
       as a Statice schema.  However, it is not.  Perhaps
       the application program is not loaded, not all
       loaded, or loaded into some other package than
       #<Package DEMO 54576345>.
```

If the schema and its package are present in the Lisp world, then the Browser sets its "current database" to that name given in the Open Database command, and indicates this in the label of the viewing window.

Using the Query Command to Select Entities

Next, you should select some entities using the Query command. The Query command accepts one argument—an entity type name. If you don't know what entity types are available in the database, pressing the HELP key after typing in the Query command will display a list of possibilities. Once an entity type is entered and the RETURN key pressed, an AVV menu with all of the attributes for that entity type will appear in the typeout window. Query restrictions for the attributes

may then be entered for any of the attributes. Once a query is entered for an attribute, another line with the same attribute name on it appears below it, so that additional restrictions may be given. If an entity type inherits attributes from other entity types, those attributes also appear in the menu.

For example, given the following entity type definitions:

```
(define-entity-type person ()
  ((name string :unique t :inverse person-named :inverse-index t)
   (ssn integer :unique t :initform 0 :read-only t)
   (picture image)))

(define-entity-type employee (person)
  ((home-address string :area emp-hom :function-set nil)
   (office string)
   (phones (set-of string))
   (salary integer)
   (dependents integer)))

(define-entity-type faculty (employee)
  ((rank string)
   (teaching (set-of course) :inverse teachers-of)
   (tenure boolean :initform nil)
   (dept department))
  (:area faculty-club)
  (:type-set nil))
```

The following AVV menu will be displayed when Query Faculty is given:

```
PERSON-NAME: Any
PERSON-SSN: Any
PERSON-PICTURE: Any
EMPLOYEE-HOME-ADDRESS: Any
EMPLOYEE-OFFICE: Any
EMPLOYEE-PHONES: Any
EMPLOYEE-SALARY: Any
EMPLOYEE-DEPENDENTS: Any
FACULTY-RANK: Any
FACULTY-TEACHING: Any
FACULTY-TENURE: Any
FACULTY-DEPT: Any
```

Entering Query Specifications

To query for all faculty which have a salary > 10 and < 100, you can click on the *Any* next to EMPLOYEE-SALARY: and enter the operator **>** and the value **10**. A new EMPLOYEE-SALARY: line with the value *Any* will then appear below into which you can enter < 100.

```
PERSON-NAME: Any
PERSON-SSN: Any
PERSON-PICTURE: Any
EMPLOYEE-HOME-ADDRESS: Any
EMPLOYEE-OFFICE: Any
EMPLOYEE-PHONES: Any
EMPLOYEE-SALARY: > 10
EMPLOYEE-SALARY: < 100
EMPLOYEE-SALARY: Any
EMPLOYEE-DEPENDENTS: Any
FACULTY-RANK: Any
FACULTY-TEACHING: Any
FACULTY-TENURE: Any
FACULTY-DEPT: Any
```

When the query is performed, these specifications are all "anded" together; the result of the query is the set of entities which match the conjunction of all specifications entered.

You may use any of the following operators for query operator specifications:

```
= ≠ < > ≤ ≥ any eq eql equal string-prefix string-prefix-exact
string-search string-search-exact string-equal string-greaterp
string-lessp string-not-greaterp string-not-lessp string=
string≠ string< string> string≤ string≥
```

Of course, not all of them may be useful depending on the attribute type; for example, **string=** should not be used on an integer attribute. The operator **any** may be used to specify that any value may be used as a match for the query. You may use entity handles as part of the query specification by clicking on them in another window (e.g. the Current Candidates window). You should use the **eq** operator for entity-valued attributes.

Finalizing a Query

Pressing `END` executes the query. During query execution, progress notes indicate a "Counting" phase where the Browser is counting the number of entities that match the query specifications, followed by an "Executing Query" phase where the entities are being gathered.

Presentation of Queries

Following query execution, a new query item is placed in the Queries pane. In the Current Candidates pane, a smaller presentation of the query object and all the entities in that query are displayed. The query object presentations look like this:

```
[Query on FACULTY (2 entities)]
```

There is an arrow pointing to this object, indicating that it is the current query. In this case, two entities were found matching the query specifications.

Selecting a Query or an Entity

Clicking Left on a query in the Query Pane selects that query and causes the first entity in that query to be displayed in the Viewing pane.

Clicking Left on any entity handle in the Current Candidates pane selects that entity and displays it as the current entity in the Viewing pane.

Stepping Through the Entitites in a Query

You can step through the entities in a query by any of the following methods:

- Use the Next Candidate command. It takes one argument (the number of candidates to step down) which defaults to 1. There is also a Previous Candidate command.
- Use the mouse to click on the entity in the Current Candidates pane.
- Use `c-N` to step down to the next entity handle in the query set. This key also accepts numeric arguments, so that you can do things like `c-U c-N`, or `c-10 c-N`. The `c-P` key steps backwards.

Appearance of Entities in the Viewing Pane

In the Viewing pane, the current entity handle is displayed with one attribute and its value(s) per line. For example, selecting a GRADUATE entity as the current candidate might cause the following to be displayed in the viewing pane:

```
#<Entity-Handle of UNDERGRADUATE 34/5 330006171>
    (type DEMO:UNDERGRADUATE)
NAME:            Joe Cool
SSN:             222000004
PICTURE:         [Click here to display image]
ADVISOR:         #<Entity-Handle of FACULTY 32/8 534375465>
MAJOR:           #<Entity-Handle of DEPARTMENT 28/13 534401532>
ENROLLMENT:      (Entity-Handle of ENROLLMENT 31/33
                  Entity-Handle of ENROLLMENT 31/32
                  Entity-Handle of ENROLLMENT 31/31 ... (1 more))
GPA:             0
YEAR             4
```

Notice that set-valued attributes are displayed as a group of entities (e.g. the ENROLLMENT attribute above) with ellipses and a count if there are more than three in the set.

You can click Right on the ellipses and select the "Display more of the set" menu item to see more of the set-valued attributes. You may also select any of the entities shown as the current entity by clicking Left on it. For example, you could view the advisor of Joe Cool above by clicking on the #<Entity-Handle of FACULTY 32/8 534375465> presentation in the above example.

Presentation of the Values of Attributes

The Browser uses the attribute's type as the presentation type when displaying values. Note that the above entity display has a presentation next to PICTURE: which may be clicked on to view the image of Joe Cool; the entity type definition is **statice:image** which is not a presentation type. The presentation type used by the Browser may be overriden by placing a **statice::browser-ptype** property on an attribute name symbol's plist. For example, if there is a presentation type called **pop-up-image**, then the following form would cause the Browser to present values of person-picture as **pop-up-image**.

```
(setf (get 'person-picture 'statice::browser-ptype) 'pop-up-image)
```

Describing an Attribute's Definition

You may click Middle on an attribute name in the left column of the Viewing pane to describe its definition. For example, clicking Middle on PICTURE: in the above example would cause the following to be typed in the typeout window:

```
DEMO:PICTURE is a slot in entity-type DEMO:PERSON with accessor
function DEMO:PERSON-PICTURE and type STATICE:IMAGE.

It is single valued, not unique, modifiable, not cached, and is
in the default area.  It does not have an inverse.
```

Modifying a Flavor Instance or an Attribute's Value

When you describe a flavor instance, you may click `c-m-Right` on any of the values in order to modify that slot.

The same is true for attribute values displayed in the Viewing pane. When you click `c-m-Right`, enter a new value, and press `RETURN`, a new transaction is opened, and the attribute value for the entity is set.

Note that the transaction is opened *after* the value is entered so that the Browser doesn't hold a lock on the pages that the entity resides on. That is, once the new value is entered, no check is made to see if the entity has been changed between the time that it was displayed in the Viewing pane and the time when the `RETURN` key is pressed. Thus, it is possible that you believe you are changing a value that is 0 to be 1, but sometime after the 0 was displayed by the Browser, another user had already modified it to be a different value.

Modifying Groups of Entities: Begin Edits

Another facility exists for changing groups of entity values in a more controlled manner. The Begin Edits command causes all subsequent changes to entities (using the `c-m-Right` mouse action) to be queued until an End Edits or Abort Edits command is given. End Edits attempts to make the changes based on a comparison check specified in the command, and Abort Edits throws out all the pending changes.

There are two options to the End Edits command:

:Compare Attributes
{None, Changed, All} Specifies which attributes of the entities that were changed should be compared for differences between when they were displayed in the Viewing Pane and their values at modification time. None causes no comparison to be done. Changed specifies that only those attributes that were changed for each entity should be compared. All specifies that all attributes for the changed entities need to be compared. The default is Changed.

:Difference Action {Query, Abort, Ignore} Specifies what action should be taken if the compared attributes turn out to be different. Query lets the user select which value should be used from the original displayed value. Abort aborts the changes. Ignore ignores the differences and writes the new values anyway. The default is Query.

Note that during the comparison phase, a separate transaction is opened to read the values prior to the modification writing. If no differences are detected, then the same transaction is used to write out the edits to the database. If changes are detected, the difference action is taken. If the :Difference Action is QUERY, the user is asked for resolution, and the comparison phase is repeated. If the values are changing "out from under" the Browser user, then they may be asked many times about the difference action (since each comparison phase is in a new transaction).

Clearing the Browser

You can clear the Browser by using the Clear command. This clears all the windows and resets the current database.

The Browser's Typeout Window

The Browser has a typeout window. If an error or other type of message should come up in that window, it can always be cleared with the SPACE key. c-m-SCROLL allows you to have the last screen full of the typeout window appear. It can be used repeatedly to scroll backwards through the window. Similarly, c-scroll can be used to scroll forward through the typeout window.

3.3. Statice Buffer Replacement

Statice supports buffer replacement, which is a scheme to avoid keeping every paged touched resident in virtual memeory. Buffer replacement allows Statice to access databases larger than virtual memory. Buffer replacement should also reduce the size of the working set of Statice applications, and thus improve performance.

Buffer replacement works by limiting the number of pages allocated in virtual memory on a particular machine (client or server), based on a policy set by the user. If a transaction attempts to allocate a page over the specified limit, the buffer replacement mechanism will attempt to recycle a page which has not been used recently, rather than allocate a new page. Only pages not currently involved in a transaction may be recycled. If all pages are currently in use by at least one transaction, then a new buffer is allocated, despite the limit. Only the number of pages specified by the users are maintained in a least-recently-used (LRU) fashion. Any pages over that limit are maintained in a most-recently-used fashion, to minimize virtual memory paging overhead. Thus, the buffer replacement mechanism permits the user to select the trade-off between LRU performance and paging performance, scaled to the amount of physical memory on the machine.

You can use **dbfs:set-buffer-replacement-parameters** to tune some buffer replacement parameters.

dbfs:set-buffer-replacement-parameters &key *(:page-pool-factor* **0.25***) (:page-pool-limit* **(* 1024 1024)***)*
Enables the user to limit the amount of virtual memory Statice will use as least-recently-used (LRU) buffer space.

3.4. Dealing with Strings in Statice

This chapter discusses the different ways that strings are compared in Statice, and then mentions some operators that are specially designed for dealing with strings.

3.4.1. Regular Comparison Versus Exact Comparison

Statice inverse reader functions on string-typed attributes find an entity whose value, for the attribute in question, equals a supplied value. But what does "equals" mean? Common Lisp has two kinds of string equality, which we call regular (default) string equality, and *exact* string equality. Regular string equality is sometimes called *case-insensitive* equality, and exact string equality is sometimes called *case-sensitive* equality.

The two kinds of equality testing are implemented by the Common Lisp functions **string-equal** and **string=**. **string-equal**, in Symbolics Common Lisp, compares two strings character by character, but ignores the case (upper or lower) and the character style (face, family, and style). **string=** pays attention to the case and character style. For example, regular equality says that "King" and "king" are equal and "queen" and "*queen*" are equal, whereas exact equality considers each pair not equal.

Statice queries also provide both kinds of equality testing. The conditions in the **:where** clause of a **statice:for-each** have a comparison symbol, such as **string-greaterp**. Statice also provides comparison symbols for the exact comparison functions, such as **string>**. For basic information about **:where** clauses: See the section "General Rules of the **:where** Clause of **statice:for-each**", page 43.

3.4.2. Exact Inverse Accessor Functions

Statice uses regular comparison by default. For example, in the Bank example (defined in the section "Basic Concepts of Statice") the inverse function **account-named** uses regular comparison, and will consider a person named "d. e. jones" to be the same as "D. E. Jones".

```
(define-entity-type account ()
  ((name string :inverse account-named :unique t)
   (balance integer)))
```

To get an inverse function that uses exact comparison, use the **:inverse-exact** attribute option. For example:

```
(define-entity-type account ()
  ((name string :inverse-exact account-named-exactly :unique t)
   (balance integer)))
```

account-named-exactly is an *exact inverse reader function*. It's like an inverse reader function, but uses exact comparison instead of regular comparison. For example, **(account-named-exactly "joe")** finds a person whose name is "joe", but does not find a person whose name is "Joe".

You can have both a regular inverse reader and an exact inverse reader for the same attribute:

```
(define-entity-type account ()
  ((name string :inverse account-named
                :inverse-exact account-named-exactly
                :unique t)
   (balance integer)))
```

The uniqueness checking on the **name** attribute always uses exact comparison.

3.4.3. Exact Indexes

Two Kinds of Indexes

When you use an inverse index on a string-typed attribute, you are implicitly doing a kind of string comparison. For example, consider the inverse index on the **name** attribute of the **person** entity type. (See the section "Introduction to Indexes in Statice", page 45.) When we call **(person-named "Joe Cool")**, Statice goes to the index to find an index entry whose key matches the string **"Joe Cool"**. This matching is a string comparison.

Like any string comparison, it could be regular or exact. Which kind of comparison does Statice use? Either kind, or both kinds, might be useful: it depends whether you're trying to speed up a regular inverse accessor, or an exact index accessor, or both.

Two different kinds of indexes are provided by Statice, one for each kind of comparison. The indexes that we've discussed so far all use regular comparison; there is also a kind that uses exact comparison. For every way to make or manipulate a regular index, there is a corresponding way to make or manipulate an exact index.

You can have both a regular index and an exact index, if you want to speed up both kinds of searches.

How to Make Exact Indexes

To make an exact inverse index on an attribute, use the attribute option **:inverse-index-exact**. For example:

```
(define-entity-type account ()
  ((name string :inverse-exact account-named-exactly
        :inverse-index-exact t :unique t)
   (balance integer)))
```

You can use both **:inverse-index** and **:inverse-index-exact** in the same attribute. You can use these only on attributes of type **string**, **limited-string**, or **symbol**.

There is no such thing as **:index-exact**, because the argument of an accessor function is never string-typed; it's always entity-typed.

To make an exact multiple index, use the **:multiple-index-exact** entity type option. You can have a regular multiple index and an exact multiple index on the same sequence of attributes, and they are considered distinct indexes. For more information on multiple indexes: See the section "Multiple Indexes", page 51.

All functions for making and deleting indexes take a keyword argument called **:exact**. The default value is **nil**, which means that the function operates on a regular index. If the value is **t**, the function operates on an exact index. The functions are **statice:make-index**, **statice:delete-index**, **statice:index-exists**, **statice:make-inverse-index**, **statice:delete-inverse-index**, and **statice:inverse-index-exists**. For more information on these functions: See the section "Making and Deleting Indexes", page 49.

Exact Searches with Regular Indexes

Regular indexes speed up regular searches, and exact indexes speed up exact searches. Exact indexes don't help regular searches at all. However, regular indexes do provide some help to exact searches.

When you ask Statice to do an exact search, and there is no exact index, but there is a regular index, Statice uses the regular index to help narrow down the field. For example, if you ask for the value of **(account-named-exactly "Joe")**, Statice uses the regular index to find a set of entities whose names are the same as **"Joe"** except for case. It finds **"joe"**, **"JOE"**, **"Joe"**, and **"jOe"**. Then, Statice checks each entity to see if its name is exactly equal to **"Joe"**. The index didn't do the whole job, but it probably did most of the work.

So if you sometimes do exact searches and sometimes do regular searches, but you don't want to pay the overhead of having two separate indexes, you can get most of the benfits of both indexes for the cost of a single index by making only a regular index.

3.4.4. Statice Operators for Dealing with Strings and Vectors

Statice offers several operators and one option for dealing with string-valued attributes without consing new strings. Two of the operators mentioned here work on vectors as well as strings.

The **:into** argument to **statice:attribute-value**
: Reads a string-valued attribute into an already existing string, thus preventing the consing of a new string.

statice:attribute-value-array-portion *entity-handle attribute from-start from-end into-array into-start*
: Reads a portion of an array-valued attribute into a target array.

statice:set-attribute-value-array-portion *entity-handle attribute start end from from-start*
: Writes from an array into a portion of an array-valued attribute.

statice:do-text-lines *(var string-valued-function-call* &key *(:delimiter* **'#\Return***) (:create-function* **'#'default-string-create-function***))* &body *body*
: Allows a program to iterate over the actual lines of a string-valued attribute, thus eliminating the need to cons one big string and break it into lines.

statice:do-text-lines* *function entity-handle attribute* &key *(:delimiter* **'#\Return***) (:create-function* **#'default-string-create-function***)*
: This is the dynamic version of **statice:do-text-lines**.

3.5. Opening and Terminating Databases

This section discusses the use of **statice:open-database**, **statice:with-current-database**, and **statice:terminate-database**. Before reading this section, you should review the section of the Tutorial that discusses the concepts of opening a database: See the section "Accessing a Statice Database", page 9.

As mentioned in that section, **statice:with-database** does four things:

1. Determines which database should be opened, based on the *pathname*.

2. Opens the database, if it's not already open.

3. Binds the specified variable to the database instance, during the execution of the body.

4. Makes this database be the current database, during the execution of the body.

Primitives Underlying statice:with-database

The function **statice:open-database** does the first two things: given a pathname, it finds and opens the database. It returns the database object.

The macro **statice:with-current-database** does the the fourth thing: it makes that object be the current database, for the (dynamic) extent of its body.

statice:open-database and **statice:with-current-database** can be thought of as the primitives underlying **statice:with-database**. In some cases, however, they can be useful on their own.

If you need to know what the current database is, you can use the **statice:current-database** function, which takes no arguments.

Dealing with Two Databases

One important use of **statice:with-current-database** is when you want to deal with two databases at the same time. Suppose we have a program that is reading data from one database, and writing it into another database. The important thing to note is that all references to databases must be to the current database. Here's a sample program fragment:

```
(defun copy-customers-to-people (from-pathname to-pathname)
  (with-database (from-db from-pathname)
    (with-database (to-db to-pathname)
      (with-transaction ()
        (with-current-database (from-db)
          (for-each ((c customer))
            (let ((name (customer-name c)))
              (with-current-database (to-db)
                (make-person :name name)))))))))
```

When we are doing the **statice:for-each** to search for customers, and when we are calling **customer-name**, **from-db** must be the current database. But when we are calling **make-person** to make the new entity, **to-db** must be the current database.

The example above also finally explains the purpose of the variables in the **statice:with-database** form, which we've never used until now.

Using statice:open-database and statice:with-current-database for Speed

Remember that a database is opened only once, and then it stays open. This means that **statice:with-database** on an already-open database is much faster than it is on a database that is not yet open. However, **statice:with-database** still does take some time, even if the database is already open. By using **statice:open-database**, and storing or keeping track of the database object yourself, you can avoid this cost, and just use **statice:with-current-database** whenever you want to operate on the database. **statice:with-current-database** is extremely fast.

Pitfalls with statice:open-database and statice:with-current-database

If this approach is faster, why ever use **statice:with-database**? The reason that **statice:with-database** still does take some time, even if the database is already open, is that **statice:with-database** checks that the pathname you specified still refers to the same database that it used to. If you use **statice:open-database** to open a database named "a", and someone renames that database to "b", your program will keep referring to the same database even though the name has changed. If someone now renames "c" to "a", your program still uses the original "a" rather than the current "a". This could be confusing.

By the way, while a transaction is running, locks are held to prevent any other transaction from renaming or deleting any database that the transaction has opened (with **statice:open-database** or **statice:with-database**). If the transaction uses a database that it did not open (in other words, it uses a database object and **statice:with-current-database**), then the database can be renamed or deleted during the transaction. If a transaction has read or written a database, and that database is deleted before the transaction ends, the transaction immediately aborts.

Terminating a Database

In the section "Accessing a Statice Database", we said that there is no need to close a database. Although this is true, there is a way to undo the effects of opening a database. It's called terminating a database; you do it with the function **statice:terminate-database**, which takes the pathname of the database as its argument. Note that, like opening a database, this does not have any effect on the persistent state of the database; it only affects state within your own Lisp environment.

Why terminate a database? One reason is to allow the entity handles associated with the database to be garbage collected. But the primary reason is more complicated:

Statice provides two ways for you to make an exact copy of a database. You can use the Copy Statice Database command, or you can do it with the backup dump system. Every Statice entity has its own unique identity, represented internally by a numerical unique ID. Now, if you have databases A and B, and B is a copy of A, you could access two distinct entities (one in each database) that had the same unique ID. Statice would get very confused, because the unique IDs would not really be unique. This cannot be allowed. Therefore, you cannot open two database at once if one of them is a copy of the other.

If A is open, and you try to open B, an error is signalled. The message is something like this:

```
The database X:>a exists and has the same
unique ID as the database in file X:>b
```

If you really want to open B, you must first terminate A. This is the purpose of **statice:terminate-database**.

When the above error is signalled, Statice provides a proceed handler that offers to terminate the database that's open. In the example, the proceed handler would offer to terminate X:>a.

3.6. Built-In Statice Types

This chapter describes all the built-in Statice types. For each type, it explains the possible values that Statice can model, and the Lisp representations of the values. It explains some of the fine points of how Statice deals with certain kinds of Lisp values; for example, the treatment of fill pointers in strings, and strings that are indirect arrays.

It also tells how much storage is occupied by each value. Some values take an integer number of (32-bit) words. Other values use up a fractional number of words: a number of bytes, or even just a few bits. Statice attempts to pack these smaller values together into words so as to minimize the total number of words allocated.

For some values, the amount of storage depends on the value itself. For example, a value of type **string** takes more space when the string is longer. This variable portion of storage is always rounded up to the nearest word.

Statice also understands Common Lisp types and presentation types that inherit from, expand into, or are equivalent to built-in Statice types. For example, **(mod 3)** and **(signed-byte 5)** can be used as Statice types.

In addition to the built-in types, you can extend Statice to use types of your own, either by defining presentation types that inherit from built-in types, or by using the **statice-type:define-value-type** special form to define new types. See the section "Defining New Statice Types", page 84.

statice:all-but-entity *Statice Type Specifier*

Represents any Lisp object *except* entity handles. The restrictions on what kinds of Lisp objects are supported are the same as noted for the **t** Statice type specifier: See the statice type specifier **t**, page 81.

The **t** Statice type specifier represents any Lisp object, including entity handles. The **statice:entity-handle** Statice type specifier represents entity handles.

The reason this type exists is to improve performance. When **statice:delete-entity** searches for all existing references to the entity being deleted, it has to examine every value that might hold such a reference. Since any entity could be stored in any attribute of type **t**, such an attribute must be checked, for every entity. This type cannot store entities, so its presence in the database does not slow down the **statice:delete-entity** operation.

statice:alist-member *Statice Type Specifier*

Represents an association list of items. **equalp** is the equality predicate. So, for example, "A" is considered to be a proper value of the following type:

```
((alist-member :alist (("a" . a) ("b" . b))))
```

See the presentation type **alist-member** in *User Interface Dictionary*.

alist-member is implemented by encoding the possible values into the smallest possible packed bit field. If the attribute does not provide the **:no-nulls** option, the null value counts as a possible value. For example, the example type above occupies two bits if **:no-nulls** is provided, and three bits if **:no-nulls** is not provided.

boolean *Statice Type Specifier*

Represents a boolean value (true or false). The Lisp representation is **nil** for false and **t** for true. Note that only the symbol **t** is accepted as a representation of true, not any non-**nil** object; this is for compatibility with the definition of the Lisp **boolean** type.

Because **nil** is the Lisp representation of a value, you can't use **nil** to mean the null value when you use accessor functions. See the section "The Statice Null Value", page 29.

The implementation of **boolean** depends on whether the attribute allows null values or not. See the section "The **:no-nulls** Attribute Option", page 31. A **boolean** value occupies one bit if null values are not allowed, or two bits if null values are allowed.

character *Statice Type Specifier*

Represents a Lisp character, in any character set, with any bits, and in any character style. The Lisp representation is the corresponding Lisp character object. See the type specifier **character** in *Symbolics Common Lisp Dictionary*. See the presentation type **character** in *User Interface Dictionary*. See the section "Type Specifiers and Type Hierarchy for Characters" in *Symbolics Common Lisp Language Concepts*.

There is one exception: the character whose code is 255 cannot be represented, because the value 255 is reserved to represent the null value. Since 255 is an unused character code within the standard character set, this is unlikely to cause problems.

A value of type **character** occupies sixteen bits. In addition, every time a new combination of character set, style, and bits is used, an entry is added to a per-database table. The size of the entry depends on the many things, such as the length of the name of the character style, etc, but is typically on the order of ten words long.

double-float *Statice Type Specifier*

Represents a floating point number, stored in IEEE "double" format (64 bits). The Lisp representation is a Lisp **double-float** value. Values must be of type **double-float**; no coercion is performed from other types, not even from **single-float**.

The Common Lisp **double-float** type accepts data arguments, to let you specify subranges. Statice understands these data arguments and does appropriate error checking. Examples:

`(double-float 2.0d0 5.0d0)`	A double-float x where $2.0 \geq x \geq 5.0$
`(double-float 2.0d0 (5.0d0))`	A double-float x where $2.0 \geq x > 5.0$
`(double-float 2.0d0 *)`	A double-float x where $2.0 \geq x$

See the type specifier **double-float** in *Symbolics Common Lisp Dictionary*.

A **double-float** value occupies two words.

statice:entity-handle *Statice Type Specifier*

Represents any type of entity handle. This occupies one word, as does any entity-typed attribute.

statice:image *Statice Type Specifier*

Represents a 2-dimensional bit array. For an array to be an image, one of the following must be true:

- The width is a multiple of 32.
- The array is an indirect array conformally displaced to an underlying array whose width is a multiple of 32.

Values of the Statice image type represent two-dimensional, black-and-white, image data. The Lisp representation of such an image is a raster, created with the function **make-raster-array**. See the section "Rasters" in *Symbolics Common Lisp Language Concepts*.

Because of restrictions of the **bitblt** function, the raster's width must be a multiple of 32. However, sometimes it is important to be able to represent images whose width is a specific number that's not a multiple of 32. The solution is to make a conformally displaced raster of the desired width, that is displaced to an underlying raster whose width is a multiple of 32. When **graphics:draw-image** is given such a conformally displaced array, it only draws a region the size of the displaced array.

The example program in the file **sys:statice;examples;image.lisp** demonstrates a simple example of using the **statice:image** type. A snapshot instance has a name and a picture; the type of the picture attribute is image. **setup-image-database** makes the database and puts two snapshots into it. The first snapshot holds a raster whose width is a multiple of 32, and the second snapshot holds a conformally displaced raster, 59 wide, displaced to the first raster. The function **show-image-database** displays all the snapshots in the database.

integer *Statice Type Specifier*

Represents any mathematical integer, without limit. The Lisp representation is a Lisp **integer** value.

The Common Lisp **integer** type accepts data arguments, to let you specify integer subranges. Statice understands these data arguments and does appropriate error checking. Examples:

```
(integer 2 5)          2, 3, 4, or 5
(integer 2 (5))        2, 3, or 4
(integer (2) *)        an integer greater than 2
```

See the type specifier **integer** in *Symbolics Common Lisp Dictionary*.

The implementation of **integer** depends on the data arguments. If there are two data arguments, and each is numeric rather than *, the type is a bounded subrange with a finite number of possible values. In this case, Statice uses a "packed" field, whose length is bits is the logarithm base 2 of the number of possible values (including the null value if **:no-nulls** is false).

Otherwise, there is an infinite range of values. In this case, the implementation alway takes at least one word. It takes exactly one word if the integer value is less than **(expt 2 30)** and greater than or equal to **(- (expt 2 30))**. If the value is outside that range, it takes more and more words, as necessary to hold the value, storing one word for each additional 31 bits needed to represent the integer.

statice:limited-string *Statice Type Specifier*

(statice:limited-string *n***)** represents a vector of characters with no more than *n* characters. A single data argument *n* must always be given. The limited-string can be any length less than or equal to *n*. Note that the meaning of the data argument to **statice:limited-string** is not the same as that of string's data argument.

The characters of the string must all be of type **string-char**, which means they must be in the standard character set with bits field of zero and style of NIL.NIL.NIL. Furthermore, character codes 128 and 255 cannot be represented. See the type specifier **string-char** in *Symbolics Common Lisp Dictionary*. See the section "Type Specifiers and Type Hierarchy for Characters" in *Symbolics Common Lisp Language Concepts*.

(statice:limited-string *n***)** values occupy exactly *n* bytes of storage, regardless of how long the string is. Any leftover space in the last (or first) word is available for other values. For example, two values of type **(statice:limited-string 6)** occupy exactly three words.

The purpose of this type is to provide high storage efficiency, in exchange for restrictions on the string. This type is similar to the string types provided in conventional relational database systems.

statice:limited-string is appropriate mainly when you know in advance that there is a particular, short limit on the length of strings that will ever appear in a field. For example, the three-letter codes used to designate airports could be stored in an attribute of type **(statice:limited-string 3)**. However, it is unwise to use **statice:limited-string** simply because you think it's *unlikely* that an attribute value will exceed a certain length. Unless you're sure, it's best to stick with **string**, rather than have to truncate or otherwise mangle a data value to make it fit into a fixed-length field.

member *Statice Type Specifier*

Represents one of a series of objects. **eql** is used to test a value for equality. So, for example, **a** is considered to be a proper value of the following type:

```
(member a b)
```

The following type is nearly useless, since other strings with the same characters are not considered to be equal to the string objects in the type:

```
(member "a" "b")
```

See the type specifier **member** in *Symbolics Common Lisp Dictionary*. **member** is implemented by encoding the possible values into the smallest possible packed bit field. If the attribute does not provide the **:no-nulls** option, the null value counts as a possible value. For example, the type **(member a b)** occupies two bits if **:no-nulls** is provided, and three bits if **:no-nulls** is not provided.

dw:member-sequence *Statice Type Specifier*

Represents one of a sequence of objects. **eql** is used for comparison. For example, **a** is considered to be a proper value of the following type:

```
(dw:member-sequence (a b))
```

Since **eql** is used for comparison, the following type is nearly useless, because other strings with the same characters are not considered to be equal to the string objects in the type.

```
(dw:member-sequence ("a" "b"))
```

You can provide the equality-testing function explicitly by using the **:test** presentation argument. For example, "A" is considered to be a proper value of the following type:

```
(dw:member-sequence ("a" "b") :test equalp)
```

When providing an equality-testing function, be sure to choose a function that can accept any Lisp argument as input, such as **equalp**, rather than one that signals an error when given certain objects, such as **string-equal**.

See the presentation type **dw:member-sequence** in *User Interface Dictionary*.

dw:member-sequence is implemented by encoding the possible values into the smallest possible packed bit field. If the attribute does not provide the **:no-nulls** option, the null value counts as a possible value. For example, the type **(dw:member-sequence (a b))** occupies two bits if **:no-nulls** is provided, and three bits if **:no-nulls** is not provided.

statice:pathname *Statice Type Specifier*

Represents a Genera pathname, from the generic file system. The Lisp representation is a pathname instance. When a logical pathname is stored, it is first translated to a physical pathname. See the section "Naming of Files" in *Program Development Utilities*.

The implementation of pathname is as a logical type built on string. The pathname's **:name-for-printing**, with the host name fully qualified (including the site name), is stored as a string, and parsed when read.

single-float *Statice Type Specifier*

Represents a floating point number, stored in IEEE "single" format (32 bits). The Lisp representation is a Lisp **single-float** value. Values must be of type **single-float**; no coercion is performed from other types, not even from **double-float**.

The Common Lisp **single-float** type accepts data arguments, to let you specify subranges. Statice understands these data arguments and does appropriate error checking. Examples:

`(single-float 2.0 5.0)`	A single-float x where $2.0 \geq x \geq 5.0$
`(single-float 2.0 (5.0))`	A single-float x where $2.0 \geq x > 5.0$
`(single-float 2.0 *)`	A single-float x where $2.0 \geq x$

See the type specifier **single-float** in *Symbolics Common Lisp Dictionary*.

A **single-float** value occupies one word.

string *Statice Type Specifier*

Represents a vector of characters, of any length. The Lisp representation is a Lisp **string** value. The characters can be in any character set and any character style.

The Common Lisp **string** type accepts one data argument, which specifies the length of the string. For example, **(string 3)** means strings whose length is exactly 3. Statice performs this type check. See the type specifier **string** in *Symbolics Common Lisp Dictionary*. See the presentation type **string** in *User Interface Dictionary*.

Statice's model of a string is as a vector of characters. Statice strings do not model fill pointers or indirect or displaced arrays. If you store a string with a fill pointer, Statice stores all of the characters up to the fill pointer, but does not take any notice of the rest of the elements or of the total size of the array. If you read a string value from Statice, you get a string with no fill pointer whose **:element-type** is **character**. To read string information from Statice into other arrays, use the special facilities for strings:

See the section "Statice Operators for Dealing with Strings and Vectors", page 72.

The implementation of **string** depends on whether the string has any non-thin characters. By thin characters, we mean characters in the standard character set whose style is NIL.NIL.NIL. A thin string of length n takes up **(ceiling** n **4)** words, plus one word, plus an extra word if the n is greater than 4095. Strings with non-thin characters are encoded using a set of algorithms that use data-compression techniques such as run-length encoding to minimize storage usage; there's no simple formula for the size, but it tends to be greater when many character sets and characters styles are used and when there are many transitions from one to the other. The data argument does not affect the implementation.

string-char *Statice Type Specifier*

Represents characters that are of the Common Lisp type **string-char**, namely characters that are in the standard character set with bits field of zero and style of NIL.NIL.NIL. The Lisp representation is the corresponding Lisp character object. See the type specifier **string-char** in *Symbolics Common Lisp Dictionary*. See the section "Type Specifiers and Type Hierarchy for Characters" in *Symbolics Common Lisp Language Concepts*.

There is one exception: the character whose code is 255 cannot be represented, because the value 255 is reserved to represent the null value. Since 255 is an unused character code within the standard character set, this is unlikely to cause problems.

A value of type **string-char** occupies eight bits.

symbol *Statice Type Specifier*

Represents a Lisp symbol from any package. Any Lisp symbol is a legal value. The print-name of the symbol can include characters with any character set and character style (although this is unusual). See the type specifier **symbol** in *Symbolics Common Lisp Dictionary*. See the presentation type **symbol** in *User Interface Dictionary*.

The Statice **symbol** type only stores the symbol's print name and the name of the symbol's package. It does not store the value, definition, or property list of the symbol. When you read a **symbol** value, Statice calls **intern** to find or create the appropriate symbol in the Lisp world. If the specified package does not already exist in the Lisp world, Statice signals **sys:package-not-found**, which has various proceed handlers you can use.

The implementation of **symbol** is similar to that of **string**. See the statice type specifier **string**, page 80.

It's somewhat more complicated because the package name must also be stored. Package names are shared in a table and referenced by integer indexes in order to save space. The formula for the size of strings is too complicated to present in full because of the complexity of the data compression, but in all cases common in practice a symbol takes up the same amount of space as a string: if the length of the print-name is *n*, the symbol takes up **(1+ (ceiling** *n* **4))** words.

For a discussion of what happens if you try to read, from a database, a symbol that resides in a package that doesn't exist in your Lisp environment: See the section "Obtaining a Symbol From a Database, When the Package is Undefined", page 60.

t *Statice Type Specifier*

Represents any type that can be dumped by the binary dumper. In fact, it does just that — invokes the binary dumper — to format the data. Thus, it should be used only when no more specific type is appropriate.

Below, we go into more detail about the kinds of objects that **t** can handle. For more information than is given here: See the section "Putting Data in Compiled Code Files" in *Program Development Utilities*.

The **t** Statice type specifier can handle:

- Any kind of number.
- Any character object, string, or symbol.
- Any array and any list, as long as all the elements are themselves things that the type **t** can handle.
- Compiled code objects (as long as any constants referred to by the object are themselves things that the type **t** can handle.) (Note that interpreted code objects can also be handled, because they are lists.)
- Generic function objects.
- Locatives to the value cell or function cell of a symbol.
- Instances of flavors, if and only if the flavor has a **:fasd-form** method.

The **t** Statice type specifier cannot handle the following, and will signal an error if you try to store one of them:

- All other locatives.
- Dynamic closures.
- Lexical closures.
- Logic variables.
- Stack groups.
- Circular or self-referential objects.

time:time-interval *Statice Type Specifier*

Represents a time interval. See the presentation type **time:time-interval** in *User Interface Dictionary*.

This type is the same as the **integer** type, except for the presentation aspects of the type (the printer, parser, and so on). See the statice type specifier **integer**, page 77.

time:time-interval-60ths *Statice Type Specifier*

Represents a time interval in 60ths of a second. See the presentation type **time:time-interval-60ths** in *User Interface Dictionary*.

This type is the same as the **integer** type, except for the presentation aspects of the type (the printer, parser, and so on). See the statice type specifier **integer**, page 77.

time:universal-time *Statice Type Specifier*

Represents a universal time. See the presentation type **time:universal-time** in *User Interface Dictionary*. This type is the same as the **integer** type, except for the presentation aspects of the type (the printer, parser, and so on). See the statice type specifier **integer**, page 77.

vector *Statice Type Specifier*

Represents a vector of integers. The type spec must specify the *element-type* data argument, which must be either **fixnum** or **(unsigned-byte** *n***)**, where *n* is a positive integer power of two less than or equal to 32. That is, *n*

must be 1, 2, 4, 8, 16, or 32. **fixnum** means the same thing as **(unsigned-byte 32)**. See the type specifier **vector** in *Symbolics Common Lisp Dictionary*.

The implementation of **vector** has two cases. If the data argument is **fixnum** or **(unsigned-byte 32)**, there are two fixed overhead words, plus one word for each element of the vector. Otherwise, there are three fixed overhead words, and the number of bits needed for each element is *n*.

Statice offers a convenient way to read or write a portion of a vector into a existing array, thus avoiding the need to cons a new one: See the function **statice:attribute-value-array-portion**, page 185. See the function **statice:set-attribute-value-array-portion**, page 215.

3.6.1. Types Not Supported by Statice

The following are some basic Symbolics Common Lisp types not supported by Statice:

- **and**
- **array**
- **atom**
- **common**
- **compiled-function**
- **cons**
- **function**
- **hash-table**
- **instance**
- **list**
- **locative**
- **not**
- **or**
- **package**
- **random-state**
- **readtable**
- **satisfies**
- **sequence**
- **simple-array**
- **simple-vector**
- **stream**
- **structure**
- **sys:dynamic-closure**
- **sys:generic-function**
- **sys:lexical-closure**
- **values**

3.7. Defining New Statice Types

The Statice type system is extensible: you can define new types. This section describes how to do so.

Several examples of definitions of new Statice types are provided in the file **sys:statice;examples;extended-types.lisp**.

3.7.1. Physical and Logical Statice Types

When you make a new type, you can make either a *physical type* or a *logical type*. The key difference between the two kinds of types is how data values are fetched from and stored into the database. A physical type does its own translation of a data value from "raw bits" into a Lisp object. A logical type relies on an underlying type to do this translation; it can impose a further translation between the Lisp values produced by the underlying type and the Lisp values that it shows to and accepts from the client program.

Example of Using Logical Types

Suppose you want a type whose possible values are "maroon", "ultramarine", and "turquoise". You want to disallow any other values. (This idea is similar to an "enumerated type" in Pascal.) You could make a logical type based on the **integer** underlying type, and provide functions to encode the three strings into the integer values 0, 1, and 2 when a value is written and to decode the number back into a string when a value is retrieved. When a Statice client stores data into the database or fetches data back, your functions are invoked to encode and decode the values.

Example of Using Physical Types

Suppose you want a type that can store rational numbers, but only those that are positive and whose numerator and denominator fit within eight bits. You specifically want to store these values in only 16 bits. Using a logical type based on the **integer** type would occupy at least 32 bits. (Here we assume there is no available physical type to build on, although in practice there might be.) You could define a physical type that can get direct access to the 16 bits, and build the rational number directly.

In general, logical types are easier to define and to debug than physical types. Also, bugs in the implementation of a physical type can destroy the integrity of the database. It's usually best to use logical types if you can, and use physical types only when necessary.

3.7.2. Defining Lisp and Statice Types

To make a new Statice type, there are two major steps.

1. Define a new Lisp presentation type.

2. Tell Statice how to store elements of this Lisp type into a database.

To define a new presentation type, use **define-presentation-type**. While you're at it, you can define a printer, a parser, and so on.

The name of your new type should be a symbol whose home package is normally the package of your own application program, in order to prevent name conflicts. We suggest not using keywords.

Presentation types can have data arguments and presentation arguments. The type of any Statice attribute is a full presentation type, with arguments. When you define a Statice type, however, you specify only the type name (the symbol), and so you are actually defining a parameterized family of types. Your handlers are given the presentation type arguments to examine. Make sure that the valid values for the data arguments and presentation arguments can all be read from printed representations. Statice signals an error if there is an attempt to create a type whose type specifier cannot be printed readably.

For example, to define the "enumerated" type mentioned in "Physical and Logical Statice Types", we could choose the type name **enumerated**. Then a sample presentation type might be **(enumerated "maroon" "ultramarine" "turquoise")**. The three strings are data arguments. This presentation type means a type whose values must be one of **string-equal** to one of those three strings.

If your new type is also a Lisp flavor, the **defflavor** form already defines a new Lisp type. However, you must also provide a **define-presentation-type** form as well. Be sure to use the **:no-deftype** option to **define-presentation-type** in this case.

3.7.3. Defining Logical Types

A logical type is build on top of an underlying type (which can be physical or logical). The logical type works by encoding one Lisp representation into another Lisp representation. Logical type are easier to implement than physical types.

We show the entire implementation of the **enumerated** type mentioned in "Physical and Logical Statice Types" and then explain each of the forms. This example is in the **statice-type** package.

```
(define-presentation-type enumerated ((&rest elements))
  :abbreviation-for `(and string (member . ,elements)))

(define-value-type enumerated
   (:format :logical)
   (:based-on integer))

(defmethod (encode-value enumerated-handler) (value)
  (position value (cdr presentation-type) :test #'string-equal))

(defmethod (decode-value enumerated-handler) (integer)
  (nth integer (cdr presentation-type)))
```

The define-presentation-type Form

The **define-presentation-type** form defines **enumerated** as a presentation type. This form is part of Genera's user interface substrate, and is not specific to Statice. It creates a new Lisp type similar to the **statice-type::member** type, but all of the elements must be strings.

The statice-type:define-value-type Special Form

The **statice-type:define-value-type** special form defines a new value type. The first subform is the type name, a symbol. The **:format** clause says that this is a logical type. The **:based-on** clause specifies the presentation type upon which this new type is based. The "based on" presentation type must be understood by Statice before the new type can be used in a database.

Name of the Flavor Representing a Type

When writing methods, the name of the flavor representing this type is formed by the following convention: the type's name followed by the suffix **-handler**. In this example, the **enumerated** type has the corresponding flavor named **enumerated-handler**. Thus, the methods specialize on **enumerated-handler**.

The statice-type:encode-value Method

The **statice-type:encode-value** method is given the type and a Lisp object that is the representation of a valid member of this type. It must return the Lisp object that represents the argument in the terms used by the underlying type. In general, each logical type is required to provide a method for **statice-type:encode-value**.

In the example, the **statice-type:encode-value** method takes the type and a string, finds the string's position in type's list of data arguments, and returns that position, which is an integer.

Statice guarantees the the Lisp object passed to the **statice-type:encode-value** method is of the specified type, so the method need not check. Methods for **statice-type:encode-value** need not be conerned with null values: if the underlying type is holding a null value, these methods are not called.

The statice-type:decode-value Method

The **statice-type:decode-value** method is given the type and a Lisp object that is the representation in the terms used by the underlying type. It must return the Lisp object that represents the argument in terms of this type. In general, each logical type is required to provide a method for **statice-type:decode-value**.

In the example, the **statice-type:decode-value** method takes the type and an integer, looks down the cdr of the type for the indexed element, and returns the string that it finds.

Methods for **statice-type:decode-value** need not be conerned with null values: if the underlying type is holding a null value, these methods are not called.

More about Encoding and Decoding Methods

When two elements of a logical type are compared to each other, for sorting or for "range" queries (that is, **:where** criteria involving comparisons), the underlying values are compared. Thus, the methods of encoding and decoding determine the equality predicate for values, as well as ordering of the values for comparison purposes and sorting purposes.

Another Example: Simple Integrity Checking

Here's an even simpler example of a logical type, called **string-without-e**. This is the same as the built-in **string** type, except that it does not allow its values to have the letter "e" in them. This example shows that logical types can be used to provide arbitrary constraints on the values of a type. This is a simple form of what is known in the database world as "integrity checking": making sure that the only data in the database is "valid", where the particular application defines the concept of being "valid".

```
(defun without-e (string)
  (not (find  #\e string :test #'char-equal)))

(define-presentation-type string-without-e ()
  :abbreviation-for '(and string (satisfies without-e)))

(define-value-type string-without-e
  (:format :logical)
  (:based-on string))
```

Note that it is not necessary to provide methods for **statice-type:encode-value** and **statice-type:decode-value**. The Lisp type system takes care of forbidding the letter "e", and the **statice-type:define-value-type** informs Statice that a **string-without-e** is stored the same way as a **statice-type::string**.

Computing the Underlying Type

In these examples, the value of the **:based-on** attribute is a constant, always exactly the same for any logical type. Sometimes, you might want to compute the based-on type, depending on the exact presentation type. For example, the **member** logical type constructs its underlying type to be an integer subrange that is just large enough to represent the integers that can arise, in order to save storage. For example:

```
(define-value-type member
   (:format :logical)
   (:based-on-function member-based-on))

(defun member-based-on (presentation-type)
  `(integer 0 (,(length (cdr presentation-type)))))
```

The **:based-on-function** clause can be used instead of the **:based-on** clause in any definition of a logical type. The clause names a function. When any particular presentation type is given to Statice that is handled by this logical type definition,

such as **(member a b c)**, the function is called with that presentation type as its argument. The function must return the presentation type of the underlying type, e.g. **(integer 0 (3))**.

3.7.4. Defining Physical Types

A physical type works by storing a Lisp representation of a value into actual bits and retrieving those bits to reconstruct a Lisp representation. It's harder to write a physical type than a logical type, but in some cases you can achieve greater speed and/or storage efficiency.

Records

All values are stored in containers called *records*. A record can be thought of as a contiguous vector of words. Each word holds a 32-bit signed integer. (Note that words are 32-bit quantities regardless of the word size of the host computer.) Every value of any type is ultimately represented in words and stored into records. There is no limit on the size of records. (They can span pages; they are not contiguous at the physical level of abstraction. Statice handles all this for you.)

A *record addressor* is a Lisp object that names a record. Several of the type handlers that are defined as part of a physical type are given record addressors as arguments. A record addressor can be read-only, or writable; Statice passes the appropriate kind of addressor to the appropriate type handlers.

You are responsible for making sure that you write only into the portion of the record assigned to your data item. Writing into the wrong part of the record can destroy the integrity of the database! Be sure to write your methods carefully, and test them thoroughly before you use your new physical type in a database with valuable contents.

There are four functions you can call on record addressors:

statice-storage:read-record-word *record-addressor index* &key *:buffer-p*
Reads the word specified by *index* from the record specified by *record-addressor*, and returns it.

statice-storage:write-record-word *record-addressor index new-value* &key *:buffer-p*
Writes *new-value* into the word specified by *index* of the record specified by *record-addressor*.

statice-storage:read-multiple-record-word *record-addressor start-index end-index* &key *:into :into-start*
Reads a contiguous subsequence of words from the record into an array on the data stack.

statice-storage:write-multiple-record-word *record-addressor start-index end-index new-value*
Writes a contiguous subsequence of words from an array on the data stack into the record.

Variable-Format and Fixed-Format

There are two kinds of physical types: variable-format and fixed-format. When a physical type is of *variable-format*, each individual value of that type can take up a different amount of storage in its record. When a physical type is of *fixed-format*, all values of that type take up exactly the same amount of space. In a variable-format type, the amount of space is always an integer number of words. In a fixed-format type, the amount of space can be any number of bits.

We discuss these further in sections"Defining a Variable-Format Physical Type" and "Defining a Fixed-Format Physical Type".

3.7.5. Defining a Variable-Format Physical Type

In this section we show the entire implementation of an example variable-format physical type called **hw-vector**, and then explain the forms one at a time. **hw-vector** is a type that holds vectors of arbitrary length, whose elements are signed 16-bit integers (that is, integers between -32768 and 32767 inclusive). **(hw-vector** *i***)** is the same, but the vector must have no more than *i* elements; *i* must be an integer. (Here "hw" as an abbreviation for "half-word".)

Lisp representations of **hw-vector** values can be vectors of any element type, as long as the element are all signed 16-bit integers. If the array has a leader, the leader is used as the length. The Statice representation does not store the Lisp element type, nor whether there was a leader or not, nor the contents of the array past the leader.

In this example, **hw-vector** is further defined such that it is not meaningful to compare two elements of type **hw-vector**, and it is not possible to build an index on an attribute of type **hw-vector**.

The basic idea of the implementation is to store two elements in each word. However, it is also necessary to store the number of elements; Statice tells us the length of our portion of the record in words, but we need to know it in half-words, which takes one bit. We use the first half-word to encode the low-order bit of the length. The real length is computed as the number of words in our portion of the record, times two, minus one for the first half-word, and minus another one if the length bit is set. (The length bit subtracts one, rather than adding one, so that we can represent a length of zero.)

Variable-format physical types need not concern themselves with the null value. Statice takes care of the representation of the null value automatically.

```
(define-presentation-type hw-vector ((&optional limit))
  :expander 'vector
  :typep ((value)
          (and (vectorp value)
               (let ((len (length value)))
                 (and (or (null limit) (≤ len limit))
                      (every #'(lambda (value)
                                 (typep value '(integer #o-100000 (#o100000))))
                             value))))))

(define-value-type hw-vector
  (:format :variable))

(defmethod (read-value hw-vector-handler) (addressor word-offset n-words)
  (let* ((first-word (read-record-word addressor word-offset))
         (length (- (* 2 n-words) 1 (ldb (byte 1 0) first-word)))
         (vector (make-array length)))
      (loop for j below length
            for right first nil then (not right)
            with word = first-word
            with i = word-offset do
        (when right
          (setq word (read-record-word addressor (incf i))))
        (let ((raw-byte (ldb (if right (byte 16 0) (byte 16 16)) word)))
          (setf (aref vector j)
                (if (zerop (ldb (byte 1 15) raw-byte))
                    raw-byte
                    (- raw-byte (expt 2 16))))))
      vector))

(defmethod (value-equal hw-vector-handler) (value addressor word-offset n-words)
  (let ((first-word (read-record-word addressor word-offset))
        (length (length value)))
      (and (= length
              (- (* 2 n-words) 1 (ldb (byte 1 0) first-word)))
           (loop for j below length
                 for right first nil then (not right)
                 with word = first-word
                 with i = word-offset do
             (when right
               (setq word (read-record-word addressor (incf i))))
             (unless (= (aref value j)
                        (ldb (if right (byte 16 0) (byte 16 16)) word))
               (return nil))
               finally (return t)))))
```

```
(defmethod (size-of-value hw-vector-handler) (value)
  (ceiling (1+ (length value)) 2))

(defmethod (write-value hw-vector-handler) (vector addressor word-offset n-words)
  (declare (ignore n-words))
  (let ((length (length vector)))
    (loop for j below length
          for right first nil then (not right)
          with word = (if (oddp length) 0 (dpb 1 (byte 1 0) 0))
          with i = word-offset do
      (when right
        (setf (read-record-word addressor i) word)
        (incf i)
        (setq word 0))
      (setq word (sys:%logdpb (aref vector j)
                              (if right (byte 16 0) (byte 16 16))
                              word))
          finally
            (setf (read-record-word addressor i) word))))
```

The define-presentation-type Form

As with logical types, the first step in making a new Statice type is to define a Lisp type. This is just an ordinary **define-presentation-type** form that defines **hw-vector** as a Lisp type. It could be enhanced with other clauses, such as **:parser** and **:printer**.

The statice-type:define-value-type Form

The **statice-type:define-value-type** special form tells Statice that **hw-vector** is to be represented in Statice databases using the physical type mechanism. The keyword symbol **:variable** means that this is a variable-format type, as opposed to fixed-format.

The statice-type:read-value Method

The **statice-type:read-value** method is given a record with a value stored in it. It must make and return the Lisp representation of that value.

The first argument is record addressor for the record. The second argument is the word-offset into the record of the first word of this value's portion of the record. The third argument is the length of the portion, in words. The record addressor is read-only (or, in any event, not guaranteed to be writable), and the method must not write into the record. The word-offset is a positive integer and the length is a non-negative integer.

The statice-type:value-equal Method

The **statice-type:value-equal** method receives exactly the same arguments as **statice-type:read-value**, and has the same restrictions, except that takes one extra argument, called *value*, at the beginning. *value* is the Lisp representation of a data value of this type, or the null value, if this type allows null values. The method should return true if the value in the record is considered equal to *value*. The null value must be considered equal only to the null value, and no other value.

This is the method's way of expressing what it considers equality to be. In the example, note that the method considers two hw-vectors the same if they have the same length and contents, but it does not care about the element type of the Lisp vector.

The statice-type:size-of-value Method

The **statice-type:size-of-value** method receives an argument that is a valid Lisp representation of a value of the type. It must return the number of words of a record that would be used to represent this value. Statice uses this method to determine how much space must be allocated to store a value into a record.

The statice-type:write-value Method

The **statice-type:write-value** method receives exactly the same arguments as **statice-type:read-value**, and has the same restrictions, except that it takes one extra argument, called *value*, at the beginning. *value* is the Lisp representation of a data value of this type. The addressor is writable, and the length argument is the same value as **statice-type:size-of-value** would return for the value (in fact, that's how Statice knew how much room to allocate). The method should write the value into the portion of the record, or write an indication that the value is null.

3.7.6. Defining a Fixed-Format Physical Type

We'll show the entire implementation of a fixed-format physical type, and then explain the forms one at a time. The type in example is called **tiny-rational**, and it stores rational numbers whose numerator and denominator are known to fit into sixteen bits.

The implementation uses a 16-bit byte field, aligned on 16-bit boundaries, which means that the field cannot be split across words. Eight bits hold the numerator, and eight bits hold the denominator.

The null value is represented by zero in the denominator field, which cannot be the representation of any valid value. Fixed-format physical types deal with the null value explicitly; this is discussed below.

```
(defun tiny-rational-p (value)
  (and (not (minusp value))
       (< (numerator value) #o400)
       (< (denominator value) #o400)))
```

```
(define-presentation-type tiny-rational ()
  :abbreviation-for '(and rational (satisfies tiny-rational-p)))

(define-value-type tiny-rational
  (:format :fixed)
  (:fixed-space 16 16))

(defmethod (read-value tiny-rational-handler) (addressor word-offset bit-offset)
  (let* ((word (read-record-word addressor word-offset))
         (num (sys:%logldb (byte 8 (+ bit-offset 8)) word))
         (den (sys:%logldb (byte 8 bit-offset) word)))
    (if (zerop den) *null-value* (/ num den))))

(defmethod (write-value tiny-rational-handler) (value addressor word-offset bit-offset)
  (let* ((word (read-record-word addressor word-offset))
         (new-16-bits (if (eq value *null-value*)
                          0
                          (sys:%logdpb (numerator value) (byte 8 8) (denominator value)))))
    (write-record-word
      addressor
      word-offset
      (sys:%logdpb new-16-bits (byte 16 bit-offset) word))))

(defmethod (value-equal tiny-rational-handler) (value addressor word-offset bit-offset)
  (let* ((word (read-record-word addressor word-offset))
         (num (sys:%logldb (byte 8 (+ bit-offset 8)) word))
         (den (sys:%logldb (byte 8 bit-offset) word)))
    (unless (zerop den)
      (and (rationalp value)
           (= num (numerator value))
           (= den (denominator value))))))

(defmethod (record-equal tiny-rational-handler)
           (addressor-1 word-offset-1 bit-offset-1 addressor-2 word-offset-2 bit-offset-2)
  (let* ((word-1 (read-record-word addressor-1 word-offset-1))
         (den-1 (sys:%logldb (byte 8 bit-offset-1) word-1))
         (word-2 (read-record-word addressor-2 word-offset-2))
         (den-2 (sys:%logldb (byte 8 bit-offset-2) word-2)))
    (and (not (zerop den-1))
         (not (zerop den-2))
         (= word-1 word-2))))

(defmethod (value-null-p tiny-rational-handler) (addressor word-offset bit-offset)
  (let* ((word (read-record-word addressor word-offset))
         (den (sys:%logldb (byte bit-offset 8) word)))
    (zerop den)))
```

How Fixed-Format Differs from Variable-Format

A fixed-format type definition is similar to a variable-format type definition, so we'll just discuss the differences. First, the **statice-type:define-value-type** special form says **:fixed** instead of **:variable**. The **statice-type:size-of-value** method is not provided, since the size is the same for every value of the type.

Fixed-format types must explicitly handle the null value. For purposes of these methods, the null value is represented by a special Lisp object that is the value of the symbol **statice-type:*null-value***. The implementor of a new fixed-format type must find a way to represent the null value in the record, and this representation must not conflict with the representation of any real value.

The :fixed-space Clause of statice-type:define-value-type

In every record that holds a value of a fixed-format type, there is a contiguous field of bits allocated. The field is of the same size in each record, regardless of the actual value being stored. This field of bits is called the *fixed space*. The entire representation of the value must fit into the fixed space.

The **:fixed-space** clause of **statice-type:define-value-type** is how you ask Statice for the amount of fixed space needed to hold a value.

The **:fixed-space** clause specifies two values that describe the fixed space. The first value is the size of the fixed space, in bits. The second value describes the alignment of the fixed space. If the second value is zero, the fixed space must be aligned on a word boundary. Otherwise, the second value should be a positive integer less than or equal to 16, and it means that the bit position of the first bit in the field must be an integer multiple of the second value.

When we say that a bit field is "contiguous", we count bits as going from lower-numbered words to higher-numbered words within a record, and from the least-significant bit to the most-significant bit within a word. For example, suppose there were a type whose **:fixed-space** clause specified 40 and 8. This would mean that the type uses 40 bits (five 8-bit bytes), aligned on a multiple-of-8 boundary. Such a field might start at bit position 8 of word 3, continue up through bit position 31 of word 3, continue at bit position 0 of word 4, and end at bit position 15 of word 4.

The **tiny-rational** example above returns 16 and 16, and so requests 16 contiguous bits, aligned either as the low half or high half of a word. Note that by requesting this alignment, we guarantee that the whole value will be stored in one word of the record, instead of being broken across two words, which simplifies the implementation slightly.

Methods for statice-type:read-value, statice-type:value-equal, and statice-type:write-value

These are the same as for variable-format handlers, except for the word-offset and bit-offset arguments. These arguments describe the location of the fixed space within the record: they refer to the first bit of the fixed space. The size of the fixed space, of course, is the same for all elements of this type, and so is not

passed as an argument. The bit-offset is a multiple of the second value of the **:fixed-space** clause.

These functions must all be prepared to deal with the null value. if **statice-type:write-value**'s argument is **statice-type:*null-value***, it must notice this and store a representation that means that the value is null. In this case, the denominator field being zero is used to represent the null value. Similarly, **statice-type:read-value** must check for this special representation, and return **statice-type:*null-value*** if it's found. If **statice-type:value-equal** finds a null value stored in the record, it must return **nil**. The argument to **statice-type:value-equal** is never **statice-type:*null-value***, so it's not necessary to check for that.

The statice-type:record-equal Method

The **statice-type:record-equal** method is given two records, and must determine whether they are equal. The first three arguments designate the first record; the second three arguments designate the second record. The method should return a true value if and only if the values stored in both records are non-null and equal to each other. Note that the rules for handling null values are not exactly analogous to those of the **statice-type:value-equal** method: if the value in either or both record is the null value, **statice-type:record-equal** must return **nil**.

The statice-type:value-null-p Method

The **statice-type:value-null-p** method takes the same arguments as **statice-type:read-value**. It returns true if and only if the value stored in the record is the null value.

The presentation-type Instance Variable

The **presentation-type** instance variable is the presentation type itself, which is either a symbol or a list. Methods can use it to examine the data arguments and presentation arguments. In our example, the presentation type does not take any data or presentation arguments, so the instance variable is not used.

3.7.7. Comparing Values of User-Defined Types

In both examples of physical types, we have defined types whose values cannot be compared with each other, to see which is greater or less than the other. If you supply methods that say how to compare values, Statice can be asked to sort relations based on attributes of this type, or to do "range queries", queries on relations that ask for all relations where the value of an attribute is greater and/or less than a given value.

This section describes how to make values comparable. We'll start with an example of how you could add comparability to the **tiny-rational** type. This example illustrates comparison for a fixed-format physical type.

```
;;; Modified define-value-type form for tiny-rational
(define-value-type tiny-rational
  (:format :fixed)
  (:comparable-p t))

(defmethod (value-compare tiny-rational-handler)
           (value addressor word-offset bit-offset)
  (let* ((word (read-record-word addressor word-offset))
         (num (sys:%logldb (byte 8 (+ bit-offset 8)) word))
         (den (sys:%logldb (byte 8 bit-offset) word)))
    (if (zerop den)
        *null-value*
        (let ((val-1 (* num (denominator value)))
              (val-2 (* den (numerator value))))
          (cond ((< val-1 val-2) :lessp)
                ((> val-1 val-2) :greaterp)
                (t :equal))))))

(defmethod (record-compare tiny-rational-handler)
           (addressor-1 word-offset-1 bit-offset-1
                        addressor-2 word-offset-2 bit-offset-2)
  (let* ((word-1 (read-record-word addressor-1 word-offset-1))
         (num-1 (sys:%logldb (byte 8 (+ bit-offset-1 8)) word-1))
         (den-1 (sys:%logldb (byte 8 bit-offset-1) word-1))
         (word-2 (read-record-word addressor-2 word-offset-2))
         (num-2 (sys:%logldb (byte 8 (+ bit-offset-2 8)) word-2))
         (den-2 (sys:%logldb (byte 8 bit-offset-2) word-2)))
    (cond ((zerop den-1)
           (if (zerop den-2) :equal :greaterp))
          ((zerop den-2) :lessp)
          (t
           (let ((val-1 (* num-1 den-2))
                 (val-2 (* num-2 den-1)))
             (cond ((< val-1 val-2) :lessp)
                   ((> val-1 val-2) :greaterp)
                   (t :equal)))))))
```

The :comparable-p Clause of statice-type:define-value-type

The **:comparable-p** clause in the **statice-type:define-value-type** special form means that the values can be compared. The default is **nil**. If the **:comparable-p** clause's value is **t**, the type must provide methods for **statice-type:value-compare** and **statice-type:record-compare**.

The statice-type:value-compare Method

The **statice-type:value-compare** generic function is given a record holding a value, and a Lisp representation of a value. It must return **:lessp** if the value stored in the record is less than the value supplied as an argument. It returns **:greaterp** if the record is greater than the value supplied as an argument, and **:equal** if the values are equal. If the value in the record is the null value, it must return the **statice-type:*null-value***. The value passed as an argument is never the null value.

The advantage of having an explicit **statice-type:value-compare** method, instead of using **statice-type:read-value** and comparing the the Lisp representations, is to avoid allocating Lisp storage (consing) to build a Lisp rational number, resulting in greater efficiency.

The statice-type:record-compare Method

The **statice-type:record-compare** generic function is given two records, each holding a value. It must return one of the symbols **:lessp**, **:greaterp**, or **:equal**, based on the comparision of the records. Null values are considered to be equal to each other, and greater than all other values. Like **statice-type:value-compare**, **statice-type:record-compare** avoids allocating Lisp storage for rational numbers.

Comparing Values of Variable-Format Physical Types

To make a variable-format physical type be comparable, you supply the **:comparable-p** clause in the same way as for fixed-format, and supply methods for **statice-type:value-compare** and **statice-type:record-compare**. The arguments taken by these two methods are the same for variable-format as for fixed-format, except that the bit-offset argument is replaced by the n-words argument. Also, as usual, methods for variable-format types need not worry about null values.

3.7.8. Flavors Representing a Statice Type

When you define a new Statice type, a flavor is defined to represent it. We call that flavor a *storage handler*. By default, the name of the storage handler flavor representing this type is formed by the following convention: the type's name followed by the suffix **-handler**. For example, when we use **statice-type:define-value-type** to define the type **enumerated**, Statice defines a flavor named **enumerated-handler**.

Sometimes you might want more control over the selection of the storage handler flavor. One reason is to avoid name conflicts; perhaps your program has a type named **enumerated**, but already has a flavor named **enumerated-handler** for some other reason.

A more interesting reason to take control over the selection of the storage handler flavor is to choose among several possible flavors, depending on the data arguments of the presentation type. For example, suppose you were making **integer** be a Statice type. (You wouldn't actually do that, because **integer** is already a Statice type.) The type **integer** includes integers of arbitrary precision, and must be repre-

sented using a variable format. However, the type **(integer 0 15)** can be stored in four bits, and so you might want to use a fixed format representation.

You might also want to select among storage handler flavors depending on whether the attribute specifies **:no-nulls**. For example, if you were making **boolean** be a Statice type, two bits would be needed if null values were allowed, but only one bit if they were not allowed.

Here's a simplified example:

```
(define-value-type integer
  (:handler-finder (data-arguments no-nulls)
   (declare (ignore no-nulls))
   (if data-arguments
       'fixed-integer-handler
       'variable-integer-handler)))

(define-handler-flavor fixed-integer-handler
  (:format :fixed)
  (:comparable-p t))

(define-handler-flavor variable-integer-handler
  (:format :variable)
  (:comparable-p t))
```

statice-type:define-value-type offers the clause **:handler-finder**, which has an argument list and a body. The function defined by the clause is called the *storage handler finder function*. It is called when a new attribute is made. Its first argument is the list of data arguments of the presentation type of the attribute. Its second argument is the no-nulls parameter of the new attribute. It must return a symbol that is the name of the storage handler flavor. When you use the **:handler-finder** clause, it should be the only clause.

statice-type:define-handler-flavor is a special form whose syntax is just like that of **statice-type:define-value-type**, except that it does not accept the **:handler-finder** clause, and the name is the name of the flavor itself rather than the name of a type. It accepts all the other kinds of clauses, such as **:built-on** and **:comparable-p**. It should only be used in conjunction with **statice-type:define-value-type** and **:handler-finder**.

3.7.9. Summary of Methods for Defining New Statice Types

The first step in defining a new Statice type is using this special form:

statice-type:define-value-type *type-name* &body *clauses*
Defines a new Statice value type.

In cases where you want greater control over the flavor or flavors that represent the type, you can use this special form:

statice-type:define-handler-flavor *handler-name* &body *clauses*

Used only in conjunction with **statice-type:define-value-type** and **:handler-finder**; this special form defines a storage handler flavor representing a new type.

We now discuss the generic functions that you specialize with methods, when defining new Statice types. Since many of the generic functions take the same arguments as one another, we first summarize the arguments.

3.7.9.1. Arguments to Methods for Defining New Statice Types

Many of the generic functions that you specialize when defining a new Statice type accept the same arguments. Depending on the job that the generic function is doing, it accepts some subset of the arguments described here.

handler The first argument to all of these generic functions is the type. This argument selects the methods. In the documentation of these generic functions, this parameter is called *handler*. When writing methods, the name of the flavor of this type is formed by the following convention: the type's name followed by the suffix **-handler**. For example, the **enumerated** type has the corresponding flavor named **enumerated-handler**. Methods should specialize on the name of the flavor, such as **enumerated-handler**.

value A Lisp object that is the representation of a valid member of this type.

addressor The record addressor for a record.

word-offset A positive integer that is the word-offset into the record of the first word of this value's portion of the record.

n-words-or-bit-offset A non-negative integer that is the length of the value's portion. This parameter has different semantics for variable-format versus fixed-format types:

Kind of Type	*n-words-or-bit-offset*
Fixed-format	bit offset
Variable-format	number of words

For a variable-format type, the number of words is the same value as **statice-type:size-of-value** would return for the value.

The generic functions that must be specialized are:

statice-type:encode-value *handler value*

Methods for logical types should return the Lisp object that represents the *value* argument in the terms used by the underlying type indicated by *handler*.

statice-type:decode-value *handler value*
Methods for logical types should return the Lisp object that represents the *value* argument in terms of the type indicated by *handler*.

statice-type:read-value *handler addressor word-offset n-words-or-bit-offset*
Methods for physical types should make and return the Lisp representation of the value indicated by the arguments.

statice-type:value-equal *handler value addressor word-offset n-words-or-bit-offset*
Methods for physical types should return true if the value in the record is considered equal to the *value* argument. The null value must be considered equal only to the null value, and no other value.

statice-type:size-of-value *handler value*
Methods for physical types should return the number of words of a record that would be used to represent this value. Statice uses this method to determine how much space must be allocated to store a value into a record.

statice-type:write-value *handler value addressor word-offset n-words-or-bit-offset*
Methods for physical types should write the *value* into the portion of the record, or write an indication that the value is null.

statice-type:record-equal *handler addressor-1 word-offset-1 n-words-or-bit-offset-1 addressor-2 word-offset-2 n-words-or-bit-offset-2*
Methods are given two records, and must determine whether they are equal. Methods should return a true value if and only if the values stored in both records are not the null value and are equal to each other.

statice-type:record-compare *handler addressor-1 word-offset-1 n-words-or-bit-offset-1 addressor-2 word-offset-2 n-words-or-bit-offset-2*
Methods for physical types that are comparable receive two records, each holding a value; they must return one of the symbols **:lessp**, **:greaterp**, or **:equal**, based on the comparison of the records.

statice-type:value-compare *handler value addressor word-offset n-words-or-bit-offset*

Methods for physical types that are comparable receive a Lisp representation of a value and a a record holding a value. They return **:lessp**, **:greaterp**, **:equal**, or **statice-type:*null-value***, depending on how record compares to the Lisp value.

3.8. Dynamic Statice Operations

The section "Tutorial Introduction to Statice" presented automatically-generated functions (such as accessors and entity constructors) and special forms such as **statice:for-each** and **statice:add-to-set**. When using any of these, you cannot control at run time which attribute, or which type, you want to operate on. For example, the syntax of the **statice:for-each** special form requires you to indicate the

type to iterate over, statically in the source code. If you want to write a program that decides at run time which type to iterate over, you cannot use **statice:for-each**. The same is true of the automatically-generated accessors and constructors because the name of the entity type is part of the name of the function.

This section introduces ordinary Lisp functions that perform the same operations as the automatically-generated functions and special forms. Because these are ordinary Lisp functions, they take arguments in the normal Lisp way, which lets you control all aspects of what the functions do at run time.

The automatically-generated functions and special forms are usually easier to read, and more expressive, when static behavior is all you need. The functions described here are somewhat more verbose, but they provide the extra power of run time control when it is needed.

3.8.1. Dynamic Statice Accessor Functions

statice:attribute-value is the all-purpose reader function. It takes two arguments: an entity handle, and the name of an attribute. The name of the attribute is a symbol, which can be either the name of the attribute, or the name of its reader function.

statice:attribute-value returns the same values as other reader functions: The first value is the value of the attribute, or **nil** if the attribute's value is null; the second value is **t** if the attribute's value is not null and **nil** if the attribute's value is null.

You can use **setf** with **statice:attribute-value** to set an attribute value, just as you can use **setf** with other reader functions.

Here is an example, based on a previous example: See the section "The Statice Null Value", page 29. The schema is the university example, presented elsewhere in full: See the section "Defining a Schema for a University", page 20.

The entity type is **person**, and the attribute is **id-number**. The value of the variable **george** is an entity handle.

```
;;; using the automatically-generated accessor
(setf (person-id-number george) 123)
(person-id-number george) => 123 and t

;;; using the all-purpose accessor
(attribute-value george 'id-number) => 123 and t
(attribute-value george 'person-id-number) => 123 and t
(setf (attribute-value george 'id-number) 72)

;;; further examples of using both kinds of accessors
;;; and of setting an attribute's value to null
(person-id-number george) => 72 and t
(setf (attribute-value george 'person-id-number) nil)
(person-id-number george) => nil and nil
(attribute-value george 'id-number) => nil and nil
```

The function **statice:set-attribute-value-to-null** takes the same arguments as **statice:attribute-value**, and sets the attribute value to be the null value.

statice:set-attribute-value-to-null works with any attribute. In contrast, using **setf** with **statice:attribute-value** with **nil** as the value sets the attribute value to null only if **nil** is not a valid Lisp representation of some value of the type. **statice:set-attribute-value-to-null** is discussed elsewhere: See the section "The Statice Null Value", page 29.

statice:inverse-attribute-value is the all-purpose inverse reader function. It takes three arguments: the name of the entity type of the attribute, the name of the attribute, and the value whose inverse is desired. For example:

```
(person-named "george")
```

is equivalent to both of the following:

```
(inverse-attribute-value 'person 'name "george")
(inverse-attribute-value 'person 'person-name "george")
```

statice:attribute-value-null-p is like **statice:attribute-value**, but instead of returning the value of the attribute, it simply returns **t** if the value is null and **nil** otherwise. **statice:attribute-value** does the same thing with its second returned value. The advantage of **statice:attribute-value-null-p** is that it is more efficient than **statice:attribute-value** if you don't care about the value. The gain in efficiency only matters for large values, such as bit-map images. **statice:attribute-value-null-p** is used rarely compared to the other functions described here.

3.8.2. Dynamic Entity Creation

statice:make-entity is the all-purpose entity constructor function. Its first argument is the name of the entity type, and the rest of its arguments are the same as the arguments to the entity constructor function of that entity type. For example:

```
(make-person :name "Beth" :id-number 23)
```

is equivalent to:

```
(make-entity 'person :name "Beth" :id-number 23)
```

For basic information about entity constructor functions: See the section "Making New Statice Entities", page 12. See the section "The **:initform** Attribute Option", page 32.

3.8.3. Dynamic Set Manipulation

statice:add-to-set* is the dynamic version of **statice:add-to-set**. Note that **statice:add-to-set*** is a function, whereas **statice:add-to-set** is a special form. **statice:add-to-set*** lets you specify the set to which the value should be added at run time instead of within the syntax of the program.

For example:

```
(add-to-set (student-courses joe-cool) english-101)
```

is equivalent to:

```
(add-to-set* joe-cool 'student-courses english-101)
```

Similarly, **statice:delete-from-set*** is the dynamic version of **statice:delete-from-set**:

```
(delete-from-set (student-courses joe-cool) english-101))
```

is equivalent to:

```
(delete-from-set* joe-cool 'student-courses english-101)
```

For basic information about **statice:add-to-set** and **statice:delete-from-set**: See the section "Set-Valued Attributes", page 22.

3.8.4. Dynamic Statice Queries

statice:for-each* is the dynamic version of **statice:for-each**. **statice:for-each*** is a function, whereas **statice:for-each** is a special form. **statice:for-each*** lets you specify at run time which entity type to iterate over, what criteria to use, and so on.

In Statice 2.0, **statice:for-each*** does not support the full set of functionality as does **statice:for-each**. See the section "Limitations on **statice:for-each*** in Statice 2.1".

The first argument to **statice:for-each*** is a function that you supply; this function is called once for every entity that **statice:for-each*** finds. The function is called on one argument, the entity handle, and its returned value is ignored. It's analogous to the body of the **statice:for-each** special form.

The second argument to **statice:for-each*** is the name of an entity type. The remaining arguments are keyword arguments. If none of the keyword arguments is used, the function is called once for each entity of the entity type.

```
(for-each ((faculty faculty))
  (push (person-name faculty) results))
```

is equivalent to:

```
(for-each* #'(lambda (faculty)
               (push (attribute-value faculty 'name) results))
           'faculty)
```

statice:for-each* has a **sys:downward-funarg** declaration on its first parameter, so you need not include a **sys:downward-function** declaration in your functional argument.

statice:for-each* accepts the keyword argument **:where**, which is analogous to the **:where** clause of the **statice:for-each** special form. **:where** lets **statice:for-each***

select a subset of the entities, based on conditions. For background: See the section "Using the **:where** Clause of **statice:for-each**", page 41. The value of the **:where** argument is a list of conditions. **statice:for-each*** calls the function once for each entity that satisifies all the conditions. Each condition is represented as a list of three elements:

(attribute-name function-name value)

attribute-name is the name of the attribute, or the name of the attribute's reader function. *function-name* is the name of the comparison operator, usually the name of a Lisp comparison function. *value* is a Lisp object of the same type as the attribute. (When the *function-name* is **typep**, *attribute-name* is not needed and should be **nil**.)

The following example shows two ways to accumulate a list of the names of all instructors whose salary is greater than the value of the variable **this-much**. Notice the use of backquote and comma to place the actual number into the condition. This example is the core of the function **show-instructors-paid-more-than** from the tutorial; See the section "Using the **:where** Clause of **statice:for-each**", page 41.

```
(for-each ((i instructor)
           (:where (> (instructor-salary i) this-much)))
  (push (person-name i) instructors))
```

is equivalent to the following:

```
(for-each* #'(lambda (i) (push (person-name i) instructors))
           'instructor
           :where `((salary > ,this-much)))
```

Here's how you could write the core of the function **show-full-instructors-after** (from the same section):

```
(for-each ((i instructor)
           (:where (and (equal "Full" (instructor-rank i))
                        (string-greaterp (person-name i) string))))
  (push (person-name i) instructors))
```

is equivalent to:

```
(for-each* #'(lambda (i) (push (person-name i) instructors))
           'instructor
           :where `((rank equal "Full")
                    (name string-greaterp ,string)))
```

statice:for-each* accepts the keyword argument **:order-by**, which is analogous to the **:order-by** clause of the **statice:for-each** special form. **:order-by** makes **statice:for-each*** call the function on the entities in a specified order. For background: See the section "Sorting Entities with the **:order-by** Clause of **statice:for-each**", page 44. The value of the **:order-by** argument is a list of alternating attribute names and direction names. The direction name must be either **ascending**

or **descending**. Here's how the core of the **show-courses-in-dept-sorted** function could be written using **statice:for-each***:

```
(for-each ((c course)
           (:where (eq (course-dept c) dept))
           (:order-by (course-title c) descending))
  (push (course-title c) titles))
```

is equivalent to:

```
(for-each* #'(lambda (c) (push (course-title c) titles))
           'course
           :where '(dept eq ,dept)
           :order-by '(course-title descending))
```

3.8.5. Dynamic Counting of Entities

The function **statice:count-entities*** returns the number of entities of a given entity type. You can provide a **:where** clause, which results in counting only those entities that satisfy the specified conditions.

3.9. Integrating Statice with a User Interface

In this section we describe two features of Statice that help you integrate a Statice program with a user interface.

3.9.1. Viewing an Arbitrary Statice Entity

You may call the function **statice:view-entity** from your own program to display an arbitrary entity in a window.

statice:view-entity *stream pathname entity-handle* &optional *(setf-function* **#'statice::make-browser-attribute-value-setf***) values* *Function*

Displays an arbitrary entity in a window.

stream	The window on which to view the entity.
pathname	Specifies the database which the entity handle is from.
entity-handle	The entity to be viewed.
setf-function	Should be a function of two arguments, an entity-handle and the name of an attribute, which returns a list suitable for **setf**'ing.

Here's an example from the university database. Suppose the value of ***university-pathname*** is the pathname of the university database, and that the value of **j** is the entity handle for the student named Joe.

```
(view-entity *standard-output* *university-pathname* j)

#<STUDENT Joe 504260721>  (type STUDENT)
NAME:               Joe
ID-NUMBER:          827
DEPT:               (Null value)
COURSES:            (Entity-Handle of COURSE 24/28
                     Entity-Handle of COURSE 24/25)
SHIRTS:             (Entity-Handle of SHIRT 25/11)
```

The default *setf-function*, **#'statice::make-browser-attribute-value-setf** is defined as:

```
(defun make-browser-attribute-value-setf (entity-handle fname)
  `(browser-attribute-value ,entity-handle ',fname))
```

This function is called to create the **:form** argument to **present** for the attribute values. If `c-m-Right` is clicked on (i.e. "Modify this structure slot"), the user interface management system does a **setf** operation, using the **:form** argument and the new value that was entered. By default, the Browser uses **#'statice::make-browser-attribute-value-setf** as its **:form** function, so that when a attribute slot is modified, the following form is evaluated:

```
(setf (browser-attribute-value <entity-handle>
            'name-of-attribute-being-modified)
      new-value-entered)
```

browser-attribute-value is not defined as a function, but **(setf browser-attribute-value)** is, and may be used to **setf** a single attribute value in an entity handle. Note that it performs its own transaction.

3.9.2. Presentation Type for Statice Types with Simple String Names

Here we discuss a predefined presentation type for Statice entities that have simple string names, and how to make use of it.

You might define an entity type whose name is more complicated than a single string attribute. For example, you might have entities named by three attributes called **name** (a string), **type** (a string), and **version** (an integer). You'd need to write a new presentation type for this, if you wanted to get proper input and output behavior, including completion. The source code for **statice-utilities:entity-named-by-string-attribute** is provided in the file **sys:statice;examples;presentation-type.lisp**, to use as a guide in writing your own, similar presentation type.

statice-utilities:entity-named-by-string-attribute `(()` `&key` *path-name type attribute restrictions*`)` *Presentation Type*

A presentation type for Statice entities that have simple string names, where the value of a single-valued, string-typed attribute of the entity is considered the name of the entity. Most applications are expected to make a presentation type that is an abbreviation for this presentation type.

The first three data arguments are all mandatory; the last one is optional.

pathname is the pathname of the database.

type is the name of the entity type.

attribute is the name of the single-valued, string-typed attribute of the type that serves to name entities.

restrictions is a list of criteria, just like the **:where** argument to **statice:for-each***, and it means that only the subset of entities that pass all of these criteria are considered to be part of the set.

For example, here is an entity type that has a simple string name, namely the person entity type from the university example:

```
(define-entity-type person ()
  ((name string :unique t :no-nulls t :cached t
         :inverse person-named :inverse-index t)
   (id-number integer :unique t :read-only t)))
```

If we want a presentation type that will prompt the user for the name of a person in the database in ***university-pathname***:

```
`((statice-utilities:entity-named-by-string-attribute)
   :pathname ,*university-pathname* :type person :attribute name)
```

Since that's rather verbose, you might want to make an abbreviation:

```
(define-presentation-type name-in-the-university (() &key pathname)
   :abbreviation-for
     `((statice-utilities:entity-named-by-string-attribute)
        :pathname ,pathname :type person :attribute name))
```

Having defined this abbreviation, you can do things like the following:

```
(accept `((name-in-the-university) :pathname ,*university-pathname*))
```

The presentation type implements completion efficiently, by using Statice queries in the database, rather than reading all the names out of the database.

These examples are in the file **sys:statice;examples;university-example.lisp**.

3.10. Integrating Object-oriented Programming with Statice

This section discusses how you can use object-oriented techniques in a Statice application. We have already shown (in section "Inheritance From Entity Types") that an entity type can be built on other entity types, in order to inherit attributes.

Every entity type is also Lisp flavor. You can define methods on entity types. You can also specify that the flavor corresponding to an entity type should have extra

instance variables, which you can then use in methods. Lastly, you can build an entity type on other flavors, which need not be entity types themselves. Some of these techniques are advanced, and they won't be needed in the majority of Statice applications.

3.10.1. Defining Methods for Entity Types

Every entity type is also a Lisp flavor, so you can define methods on entity type flavors.

```
(defmethod (student-mean-shirt-size student) ()
  (let ((total-size 0)
        (n-shirts 0))
    (for-each ((sh (student-shirts self)))
      (incf total-size (shirt-size sh))
      (incf n-shirts))
    (/ total-size n-shirts)))
```

student-mean-shirt-size takes one argument, a student—that is, a student entity handle, which is an instance of the **student** flavor. In the body of the method, the value of **self** is the student. The caller must call this function from within a transaction, since the function accesses the database.

The rules of flavor method inheritance apply in the usual way to both Statice and Lisp, so **student-mean-shirt-size** can be applied to **graduate-student** entity handles as well.

A particularly useful generic function to handle is **sys:print-self**. Here's an example of a **sys:print-self** method:

```
(defmethod (sys:print-self person) (stream print-depth slashify-p)
  (declare (ignore print-depth))
  (if slashify-p
      (sys:printing-random-object (self stream :typep)
        (princ (person-name self) stream))
      (princ (person-name self) stream)))
```

This method works because the **name** attribute of **person** is a **:cached** attribute. **sys:print-self** is likely to be called from outside transactions, and if **name** were not cached, the calls to **person-name** from outside a transaction would signal an error.

When defining your own methods, remember that the attributes are not instance variables. You have to use the accessor functions to get or set the attribute values. This is true even for cached attributes; although instance variables exist, you cannot access them directly. The style of using accessor functions from within method bodies may seem unusual to programmers accustomed to Flavors; however, the style is consonant with the Common Lisp Object System (CLOS).

3.10.2. Specifying Instance Variables for an Entity Handle

In the **statice:define-entity-type** form, you can specify instance variables to be included in the flavor being defined to represent the entity type. To do so, use the **:instance-variables** option. You can initialize, read, and write these instance variables (if you use the **:initform**, **:reader**, and **:writer** options for the instance variables). However, the values of these instance variables are maintained only in virtual memory, and are not stored in the Statice database. This advanced option can be used for customized caching schemes or for other purposes.

See the special form **statice:define-entity-type**, page 187.

3.10.3. Mixing Flavors Into a Statice Entity Definition

In the **statice:define-entity-type** form, the name of the new entity is the first argument. The second argument is a list of entity types from which this new type will be built. You can think of them as component types, or parent types, of the new type.

You can include in this list of component types the names of flavors that are not necessarily entity type flavors. If you do so, the normal flavor inheritance rules apply. Thus, the new flavor that is being defined to represent the new entity type will inherit from all the flavors in this list.

The result is that any methods you have defined on the component flavors are inherited by the entity type flavor. Instance variables are inherited as well.

3.10.4. Statice and CLOS

Genera 8.1 includes CLOS (Common Lisp Object System). Statice users might be interested in how Statice and CLOS interact. In general, there is no direct integration between Statice 2.1 and Genera 8.1. However, using Statice and CLOS in the same Lisp world works, and you can develop programs that use both Statice and CLOS.

In Genera 8.1 and Statice 2.1, Statice entity handles are implemented as Flavors instances, not CLOS instances. Statice attribute accessing functions are Flavors generic functions, not CLOS generic functions. You cannot define methods for user-defined CLOS generic functions that are specialized on a flavor (such as a Statice entity flavor).

Statice users can use CLOS, noting the following restrictions. You cannot mix CLOS classes into a Statice entity definition. See the section "Mixing Flavors Into a Statice Entity Definition", page 109. You cannot define CLOS classes that inherit from Statice entity flavors, or define CLOS methods that specialize on Statice entity flavors. We anticipate that some of these restrictions will be lifted in a future release of Genera.

3.11. Examining the Schema of a Statice Database

Some Statice applications are written to work with one schema that is designed specifically for that application. The univerity example from the tutorial is such an application. Other applications are written so that they can work with arbitrary schemas, so that the programmer does not know, at the time the program is being written, what entity types and attributes exist. A general-purpose browser, or a sharable interactive display and query facility, are examples.

Applications of the second kind need a way to examine a Statice schema under program control; that is, at run time. This section introduces a set of functions that let an application examine a schema.

3.11.1. Template Schemas and Real Schemas

There are two kinds of schemas in Statice: *template schemas* and *real schemas*.

The template schema is what **statice:define-schema** and **statice:define-entity-type** creates. It is independent of any particular database; indeed, immediately after it is defined, but before **statice:make-database** is called, there is a template schema even though there is no database. The template schema lives in Lisp virtual memory. It can change over time if the **statice:define-schema** and **statice:define-entity-type** forms are edited and compiled again.

The *real* schema is created by **statice:make-database** by copying a template schema. Every real schema resides in one Statice database, and describes what is in that database. Although a real schema starts out as a copy of a template schema, it can change over time. For example, **statice:make-index** and **statice:delete-index** change the real schema of one database, but do not have any effect on the template schema.

The same set of Lisp functions can examine both template schemas and real schemas. The only functions that work differently for the two kinds of schema are the ones that you start with, namely:

statice:get-real-schema *pathname*
: Returns an instance representing the real schema in the Statice database stored in the file indicated by *pathname*, which is a database pathname.

statice:get-template-schema *schema-name*
: Returns an instance representing the template schema named *schema-name*.

statice:get-template-entity-type *entity-type-name*
: Returns the entity type instance corresponding to *entity-type-name*, a symbol. (There is a separate function for getting a template entity type because template entity types can exist independent of any schema. As long as there is a **statice:define-entity-type** form, the entity type is defined, even if it's not a member of any schema.)

There is also a user interface for looking at the real schema of a particular database. See the section "Show Database Schema Command", page 171.

3.11.2. Example of Schema Examination

To show how the schema examination functions are used, we present a sample interactive session with comments. In practice, the schema examination functions are usually used by programs, rather than directly by the user. We present this example in the form of an interactive session because this is a clear way to show the functions in action, not because they're usually used interactively.

This example is based on the university schema: See the section "Defining a Schema for a University", page 20.

The schema examination functions are centered around four kind of instances that represent the fundamental objects that make up schema information: schemas, entity types, attributes, and multiple indexes. The functions let you examine the relationships among, and the properties of these objects.

We start by examining the template schema for **university**:

```
(setq u (get-template-schema 'university))
                    => #<Schema UNIVERSITY 532342233>
```

The functions **statice:schema-name** and **statice:schema-types** return the name and the entity types of a schema:

```
(schema-name u) => UNIVERSITY

(schema-types u) =>
    (#<Entity-Type PERSON 532342320>
     #<Entity-Type STUDENT 540046771>
     #<Entity-Type GRADUATE-STUDENT 532343340>
     #<Entity-Type SHIRT 540045763>
     #<Entity-Type COURSE 532343667>
     #<Entity-Type INSTRUCTOR 532343103>
     #<Entity-Type DEPARTMENT 532344047>)
```

Now let's take a look at the **student** entity type. We can obtain an entity-type instance either from the list that we just got from **statice:schema-types**, or by calling **statice:get-template-entity-type**. Once we have the entity-type instance, we can find the name of the entity type and the names of its parents:

```
(setq s (get-template-entity-type 'student))
                    => #<Entity-Type STUDENT 540046771>
(type-name s) => STUDENT
(type-parent-names s) => (PERSON)
```

We can also find out about **student**'s attributes. We'll find out about the **shirts** attribute. First, we set the variable **ss** to the attribute instance for the **shirts** attribute.

```
(type-attributes s)
   =>    (#<Attribute SHIRTS of STUDENT 540046734>
          #<Attribute COURSES of STUDENT 540047027>
          #<Attribute DEPT of STUDENT 540047107>)
(setq ss (first *)) => #<Attribute SHIRTS of STUDENT 540046734>
```

Now we can call functions that tell us the name of the attribute, the name of the reader function and the inverse reader function, the entity type that "owns" the attribute, the type of values of the attribute, whether the attribute is set-valued, and whether the attribute was specified with the **:no-nulls** option:

```
(attribute-name ss) => SHIRTS
(attribute-function-name ss) => STUDENT-SHIRTS
(attribute-inverse-function-name ss) => SHIRT-OWNER
(attribute-type ss) => #<Entity-Type STUDENT 540046771>
(attribute-value-type ss) => #<Entity-Type SHIRT 540045763>
(attribute-value-is-set ss) => T
(attribute-no-nulls ss) => NIL
```

3.11.3. Summary of Functions for Examining a Schema

Getting a Schema Instance

The first step in examining a schema is getting a schema instance, which is an instance of either a real schema or a template schema.

statice:get-real-schema *pathname*
: Returns an instance representing the real schema in the Statice database stored in the file indicated by *pathname*, which is a database pathname.

statice:get-template-schema *schema-name*
: Returns an instance representing the template schema named *schema-name*.

If you need only the name of the schema for a particular database, it's not necessary to get a schema instance; instead, use the following function:

statice:get-real-schema-name *pathname*
: Returns the name of the real schema in the Statice database stored in the file indicated by *pathname*, which is a database pathname. The result is a symbol, not a schema instance.

You can get a template entity type instance by using the following function:

statice:get-template-entity-type *entity-type-name*
: Returns the entity type instance corresponding to *entity-type-name*, a symbol. (There is a separate function for getting a template entity type because template entity types can exist independent of any schema. As long as there is a **statice:define-entity-type** form, the entity type is defined, even if it's not a member of any schema.)

Operations on Schema Instances

statice:schema-name *schema*
Returns the name of the given *schema*.

statice:schema-types *schema*
Returns a list of entity types of the given *schema*.

Operations on Entity Type Instances

You get entity type instances by using **statice:schema-types**, or **statice:get-template-entity-type**. The following operations can be used on entity type instances:

statice:type-name *entity-type*
Returns the symbol that is the name of the given *entity-type*.

statice:type-parent-names *entity-type*
Returns the names of the parent types of the given *entity-type*.

statice:type-attributes *entity-type*
Returns the names of the attributes of the given *entity-type*.

statice:type-area-name *entity-type*
Returns the name of the area in which entities of the given *entity-type* are stored.

statice:type-set-exists *entity-type*
Returns true if a set exists for the given *entity-type*; otherwise, returns **nil**.

statice:type-multiple-indexes *entity-type*
Returns a list of multiple index instances of the given *entity-type*.

Operations on Attribute Instances

You get attribute instances by using **statice:type-attributes**. The following operations can be used on attribute instances:

statice:attribute-name *attribute*
Returns the name of the given *attribute*.

statice:attribute-function-name *attribute*
Returns the name of the reader function for the given *attribute*.

statice:attribute-type *attribute*
Returns the entity type of the given *attribute*.

statice:attribute-value-type *attribute*
Returns the value type of the given *attribute*.

statice:attribute-value-is-set *attribute*
Returns true if the attribute is set-valued; otherwise, returns **nil**.

statice:attribute-unique *attribute*
Returns true if the attribute's value is defined to be unique; otherwise, returns **nil**.

statice:attribute-read-only *attribute*
Returns true if the attribute is defined to be read-only; otherwise, returns **nil**.

statice:attribute-area-name *attribute*
Returns the name of the area (a symbol) in which values of the given *attribute* are stored.

statice:attribute-set-exists *attribute*
Returns true if a set exists for the given *attribute*; otherwise, returns **nil**.

statice:attribute-index-exists *attribute*
Returns true if an index exists for the given *attribute*; otherwise, returns **nil**.

statice:attribute-index-average-size *attribute*
Returns the average size defined for the attribute's index, or **nil** if no average size was specifed for the index.

statice:attribute-inverse-index-exists *attribute*
Returns true if an inverse index exists for the given *attribute*; otherwise, returns **nil**.

statice:attribute-inverse-index-exact-exists *attribute*
Returns true if an inverse exact index exists for the given *attribute*; otherwise, returns **nil**.

statice:attribute-inverse-index-average-size *attribute*
Returns the average size defined for the attribute's index, or **nil** if no average size was specifed for the index.

statice:attribute-no-nulls *attribute*
Returns true if **:no-nulls t** was specified for the given *attribute*; otherwise, returns **nil**.

statice:attribute-inverse-function-name *attribute*
Returns the name (a symbol) of the inverse function for the given *attribute*, or **nil** if the *attribute* has no inverse function.

Operations on Multiple Indexes

You get multiple index instances by using **statice:type-multiple-indexes**. The following operations can be used on multiple index instances:

statice:multiple-index-attribute-names *multiple-index*
Returns a list of names (symbols) of the attributes indexed by this multiple index.

statice:multiple-index-unique *multiple-index*
Returns true if the *multiple-index* is defined to be unique; otherwise, returns **nil**.

statice:multiple-index-case-sensitive *multiple-index*
Returns true if the *multiple-index* is defined to be case sensitive; otherwise, returns **nil**.

3.12. Modifying a Statice Schema

Sometimes after you've created a database, you want to make a change to its schema, preferably without changing or losing the contents of the database. This is done in two steps:

1. Modify the template schema by editing and recompiling the schema definition.

2. Modify the real schema of the particular database(s) you want to update, by using the Update Database Schema command.

Statice's ability to update schemas is limited: some changes to the schema cannot be performed without losing some of the the data in the database. For information on which kind of modifications can and cannot be accomplished: See the section "Limitations to Modifying a Real Schema", page 116.

To understand the following explanation, you should be familiar with the terms *template schema* and *real schema*: See the section "Template Schemas and Real Schemas", page 110.

3.12.1. Modifying the Template Schema

Modifying the template schema is easy: you can edit the code and recompile, just as if you were updating the definition of a Lisp function. Now the template schema in the Lisp world incorporates your changes.

Here is an example based on the **university** schema. (The entire schema is presented elsewhere: See the section "Defining a Schema for a University", page 20.) Suppose you want to add a new attribute called **age** to the **person** entity type. The attribute should be of type **single-float**, and there should be an inverse index. To do this, you edit the **define-entity-type** form for **person**, as follows:

Before:

```
(define-entity-type person ()
  ((name string :unique t :no-nulls t :cached t
         :inverse person-named :inverse-index t)
   (id-number integer :unique t :read-only t)))
```

After:

```
(define-entity-type person ()
  ((name string :unique t :no-nulls t :cached t
         :inverse person-named :inverse-index t)
   (age single-float :inverse-index t)
   (id-number integer :unique t :read-only t)))
```

Then you must compile the **statice:define-entity-type** form. You can do this in the Lisp world, perhaps by using the `c-sh-C` command in Zmacs, or you can write out the file, compile it, and load it. Any technique that works for modifying any Lisp definition will also work for **statice:define-entity-type**.

3.12.2. Modifying the Real Schema

To modify the real schema of a particular database, you use the Update Database Schema command. The command takes one argument: the pathname of the database. If you have several databases, all of which were made from the same template schema, you must run Update Database Schema separately on each database.

Update Database Schema examines the template schema in the Lisp world, and compares it to the real schema in the database. It composes a sequence of operations to perform on the database in order to change its schema to correspond to the current template schema. It shows you the list of operations it intends to perform, and asks whether to go ahead. You should examine the list and make sure that it's what you want; if so, type "yes" and it actually performs the operations.

Continuing with our example in the last section, here's what the Update Database Schema command does when asked to update the database BEET:>university:

```
:Update Database Schema (pathname of database) BEET:>university
Plan for updating the schema of BEET:>university:
Add attribute AGE of entity type PERSON
Go ahead? (Yes or No) Yes
Done.
```

Some changes to the template schema don't require any changes to the real schema. For example, if you add an inverse accessor to an entity type, or remove one, all you need to do is update the template schema. The real schema need not be changed. This is because an inverse accessor is just a Lisp function, and doesn't change the layout of data in the database. There is no harm in doing Update Database Schema in such a case; it just tells you that no changes are needed.

3.12.3. Limitations to Modifying a Real Schema

When the Update Database Schema command encounters differences between the template schema to the real schema, it classifies the change required to rectify the difference as either compatible or incompatible. Compatible changes don't disturb the database; incompatible changes generally lose some data.

For example, Statice cannot change the type of an existing attribute. If the template schema says that the type of the **salary** attribute is **single-float**, but the real

schema says that the type is **integer**, the only way to update the real schema is to delete the **salary** attribute and make a new **salary** attribute. The unfortunate side-effect is that all the **salary** values disappear, and the values of the new **salary** attribute are all the null value, as if you had added a new attribute.

This is why Update Database Schema shows you what it intends to do, and waits for confirmation before actually doing it. When Update Database Schema finds it necessary to perform an incompatible operation, it tells you exactly what it plans to do. You can then decide whether to go ahead.

Update Database Schema has the following limitations:

- When an entity type is removed, all its subtypes are removed. If you remove a type from the template schema, but leave a subtype (changing the subtype's parent list, of course), Update Database Schema must delete and re-create the subtype.

- If a type's list of parents has changed, or the type's area name has changed, Update Database Schema must delete and re-create the type.

- If an attribute's type changes, if it changes whether it's set-valued or not, if it changes whether it's unique or not, if it changes whether it allows null values or not, or if it changes its area name, Update Database Schema must delete and re-create the attribute.

If none of the limitations noted above apply, then Update Database Schema can make the following changes compatibly:

- Add or delete entity types.

- Add or delete attributes.

- Add or delete any kind of index.

- Add or delete type sets and function sets.

When an entity type is deleted, all of the entities of that type (and its subtypes) are deleted, just as if **statice:delete-entity** had been called. **statice:delete-entity** can affect attribute values of other entities. See the section "Deleting Entities".

When a new attribute is added, the value of the attribute in all existing entities of the relevant entity types is the null value if the attribute is single-valued, or the empty set if the attribute is set-valued.

When an attribute is deleted, the storage occupied by the attribute values is not freed up. One way to free the storage is to use the high-level dumper and loader to rebuild the database. See the section "High-level Dumper/Loader of Statice Databases", page 163.

4. Statice Performance Issues

When a database grows large, the performance of Statice becomes slower. Statice provides techniques for increasing performance: adding indexes; and controlling the use of type sets, attribute sets, and areas. To use these techniques effectively, you need to understand some aspects of how Statice works internally, and how it represents information in a database. This section describes the implementation of Statice to the extent needed to increase performance of Statice programs.

4.1. Statice Records

Statice stores information in chunks of memory called *records*. Some records are only a few words long. Records can get arbitrarily large if, for example, they are holding long character strings or images.

The following examples are based on the schema definition in "Defining a Schema for a University".

There are two kinds of records: *entity records* and *relationship records*.

Entity Records

There is one entity record for every entity in the database. The entity record represents the entity itself, and also stores all of the values of the single-valued functions of the entity. This includes the single-valued functions that it inherits from its parent domains, as well as its own single-valued functions.

For example, there is one entity record for every student represented in the database. This record holds the values of the **student-dept** function, the **person-name** function, and the **person-id-number** function, for that student.

Relationship Record

There is one relationship record for every element of the value of every set-valued function in the database. In other words, the value of a set-valued function is stored in many relationship records, one for each member of the set.

For example, the values of the **student-courses** function are represented by relationship records. There is one relationship record for every pair of a student, and a course being taken by the student. Each relationship record, in other words, represents a fact like "Student Fred is taking course english-1". If Fred is taking english-1, math-2, and history-3, there will be three relationship records.

One-to-Many, Entity-valued Functions are an Exception

There is an exception to the above rules for one-to-many, entity-valued functions; that is, functions with the following characteristics:

- The function is unique; that is, **:unique t** was specified
- The function is set-valued; that is, its type is **(set-of** *something***)**
- The function is entity-typed; that is, *something* is an entity type

The values of one-to-many, entity-valued functions are stored inside the entity records of the entities that are the value of the function, rather than in the entity record of the entity itself, and rather than in relationship records.

For example, consider the one-to-many function **student-shirts**. Although this is a function of the **student** type, the values are actually stored in the **shirt** entity records. You can think of it like slots in a **defstruct**: every **shirt** entity record has a slot that points to the student who owns this shirt.

Another example of a one-to-many, entity-valued function is the **students-in-dept** inverse function. **students-in-dept** is a function of a department, but the information is stored in the entity records for students. That is not surprising, since **students-in-dept** is really just another way of thinking of the **student-dept** function.

Exceptions

Later on, when we discuss areas, we'll see some modifications and exceptions to these rules. See the section "Statice Type Sets, Attribute Sets, and Areas", page 127.

4.2. Statice Indexes

What is an Index

An index is a kind of physical data structure in a Statice database. An index has no effect on the semantics of a database (that is, it does not change what functions are defined, their arguments or return values, and so on). The only effect of an index is to change the speed of some operations.

The Effect of an Index

To illustrate what indexes are for, consider the function **show-students-courses**. The **statice:for-each** is given a student, and it must find all the courses that the student is taking. In other words, it finds the value of the **student-courses** set-valued function for a particular student.

The values of the **student-courses** function are represented by relationship records. To find all the courses being taken by student Fred, Statice has to find all of the **student-courses** relationship records for which Fred is the student. Then, it can find the course entities pointed to by those records.

If there is no index, the query must look at every relationship record for the **student-courses** function, searching for those records that point to Fred. If there are

a thousand students, each taking four courses, Statice must search through four thousand records. In this query, only a few of those records are part of the answer. This method of processing a query is called a *type set scan*, and it can be very slow if the database is large.

If there is an index on the function **student-courses**, the query is processed using an *index scan*. The index has a pointer from the entity record for Fred back to all the **student-course** relationship records that point to Fred as the student. By consulting the index, the query can find all the desired records directly, without bothering with all the irrelevant records. This is much faster.

The Cost of an Index

Why not use indexes on every set-valued function? The index has several costs. First, insertion and deletion in the **student-courses** sets is slower when an index is present, because the pointers in the index must be updated. Second, the index is a storage structure that takes up additional disk space. Third, the index itself is information that has to be read from the database, and reading information takes time. Fourth, if two users try to concurrently create or delete an element of any **student-courses** set, they would both try to alter the same index at the same time, and one would have to wait.

Guidelines for Using Indexes

Deciding whether to have an index or not is a trade-off. The index makes querying faster, but imposes a penalty on insertion and deletion.

We recommend using indexes when you expect that queries that will happen relatively frequently, and for functions that are queried more often than they are modified.

Fortunately, it's easy to experiment, and change your mind later. You can create and delete indexes any time you want, and compare the performance before and after.

Creating and Destroying Indexes

Usually you specify indexes as part of the schema definition, in options in the clauses of **statice:define-entity-type** forms. When the schema definition specifies indexes, **statice:make-database** creates the indexes when it creates the database. To do this, provide the **:index** option with the value **t** in the clause for the attribute.

```
(define-entity-type student (person)
  ((dept department :inverse students-in-dept)
   (courses (set-of course) :index t)))
```

You can create or delete an index at any time during a transaction by using **statice:make-index** or **statice:delete-index**, which take the name of the function as their sole argument. You can use **statice:index-exists** to see whether an index currently exists.

Inverse Indexes

You can also make indexes for inverse functions.

You can specify an inverse index in the **statice:define-entity-type** form. The following example defines an inverse function called **students-in-dept**, and uses the **:inverse-index** option to give it an index.

```
(define-entity-type student (person)
  ((dept department :inverse students-in-dept :inverse-index t)
   (courses (set-of course) :index t)))
```

This inverse index speeds up queries of the form "What students are in this department?" This includes queries that use the inverse function explicitly (by calling **students-in-dept**), and queries that use **statice:for-each** instead of the inverse function.

The function **show-students-in-english-department-1** is one example of a query that will be speeded up by this inverse index.

```
(defun show-students-in-english-department-1 ()
  (with-database (db *university-pathname*)
    (princ
      (with-transaction ()
        (with-output-to-string (string-stream)
          (for-each
            (s (students-in-dept (department-named "English")))
            (format string-stream "~%~A" (person-name s)))))))
  nil)
```

Note that the values of the **students-in-dept** attributes are stored in the entity records of type **student**. This shows that indexes can be used for information in entity records as well as for information in relationship records.

There are functions for making indexes for inverse functions: **statice:make-inverse-index**, **statice:delete-inverse-index**, and **statice:inverse-index-exists**. The argument given to these functions is the name of the accessor function, not the inverse accessor function.

Note: although inverse functions are only a syntactic convenience feature, inverse indexes are really something different from regular indexes, whose effect cannot be achieved any other way. An index and an inverse index are distinct physical structures, arranged for quite different purposes.

An Inverse Index without an Inverse Function

You can make an inverse index on a function even if there is no inverse function specified in the **statice:define-entity-type**. The following example defines the **student** type to have an inverse index, but no inverse function.

```
(define-entity-type student (person)
  ((dept department :inverse-index t)
   (courses (set-of course) :index t)))
```

(Note that simply redefining the **student** type has no effect on the database, unless you remake it or use the Update Database Schema command. See the section "Modifying a Statice Schema", page 115.

Why would you want an inverse index without an inverse function? Consider the function **show-students-in-english-department-2**, which does the same thing as **show-students-in-english-department-1**. The only difference is that it uses the function **student-dept**, instead of the inverse function **students-in-dept**. This query is speeded up by the inverse index.

Remember that inverse functions are really just syntactic sugar. Statice treats both versions of the function the same way, and uses an index scan over the entity records of the **student** type in both cases.

```
(defun show-students-in-english-department-2 ()
  (with-database (db *university-pathname*)
    (princ
      (with-transaction ()
        (with-output-to-string (string-stream)
          (for-each
            (s student
               (:where (eq (student-dept s)
                           (department-named "English"))))
          (format string-stream "~%~A" (person-name s)))))))
  nil)
```

Indexes on Other Kinds of Functions

So far, we've only used indexes with queries that were testing the equality of entities. In general, an index can be created on any function, not just an entity-valued function. It can speed up any equality test, any inequality test (numbers or strings), and any string-prefix test in a **statice:for-each** query.

For example, a function that is sped up by an inverse index is **person-named**, the inverse function of the **name** attribute of the **person** type. If there is no inverse index, **person-named** must use a type set scan, scanning over all **person** entity records, looking for one whose **name** attribute has the specified value. If there's an inverse index, Statice maintains a physical structure called a *B-tree index* that contains **name** values and pointers to the entity records containing those values. **person-named** then uses an index scan, searching very efficiently through the B-tree index to find the target entity record.

Earlier we saw the function **show-instructors-paid-more-than**. See the section "Using the **:where** Clause of **statice:for-each**", page 41. It uses the single-valued function **instructor-salary**, whose values are stored in the entity records for type **instructor**. If there is an index on this function, Statice can quickly find all the

entity records for type **instructor** that contain a salary greater than a given amount, or between any two values. This is fast because the entries that make up the B-tree index are stored in sorted order.

Case Sensitivity of Inverse Indexes

There are two kinds of indexes for accelerating the queries made by inverse functions whose value is a string. One kind of index is case-sensitive, and the other is case-insensitive. It is possible to have both kinds of index for a string-valued inverse function.

Statice uses a case-insensitive index to speed up a case-sensitive query, by narrowing down the search set, but a case-sensitive index is still faster than a case-insensitive index if there are many different strings that are the same except for case.

When specifying the inverse index in the **statice:define-entity-type** form, use the **:inverse-index** option to make an index that does not distinguish case. Use the **:inverse-index-exact** option to make an inverse index that does distinguish by case.

When using **statice:make-inverse-index**, the value of the **:exact** keyword controls whether the index is case-sensitive.

For more information: See the section "Dealing with Strings in Statice", page 69.

Indexes on Queries with Several Criteria

Indexes can also speed up queries in which several criteria are tested, and combined with an **:and** in the **:where** clause. For example, consider the function **show-full-instructors-after**, which was described in "Using the **:where** Clause of **statice:for-each**". Here's the **statice:for-each** clause from **show-full-instructors-after**:

```
(for-each ((i instructor)
           (:where (and (equal "Full" (instructor-rank i))
                        (string-greaterp (person-name i) string))))
  (push (person-name i) instructors))
```

There are two criteria that an instructor must meet, each of which is represented by one subform of the **and**: the instructor's rank must be **"Full"**, and the instructor's name must alphabetically follow **string**. To execute this query, Statice checks both criteria and looks for an index that can be used.

- If neither criterion can be resolved using an index, it just scans over **instructor**, and so must examine every entity record for the **instructor** type.

- If one of the criteria can be resolved using an index, Statice does an index scan to fetch the subset of records that fit this criterion, and then examines each record to see whether it fits the other criterion.

- If both criteria can be resolved using indexes, Statice does two index scans, one for each index. Then it does a "set intersection" on the results of the two index scans, to fetch only those records that are pointed to by both indexes.

The last way, with two index scans, is the fastest, especially when the database is large. To get this behavior for the function **show-students-in-english-department-1** or **-2**, you need two indexes: an inverse index on **student-dept**, and an inverse index on **department-name**.

Why are these both inverse indexes, rather than regular indexes? Here's how to think of it. **student-dept** is a function of a student that finds a department. But in this query, we are trying to find a student, not a department. So we're going in the "inverse" direction. Similarly, **department-name** is a function that you use when you already have a department and you want to get a name, but in this query we are trying to get a department; again, this is the "inverse" direction.

Sometimes an Index is Useless

There are some functions for which an index is useless. For example, there is no reason to make an index for the **person-name** function. Given a person entity, the person's name is easy to get: it is stored in the entity record. To be precise, it is useless to make an index on any one-to-one function or any many-to-one function, and it is useless to make an inverse index on any one-to-many entity-typed function.

It is not useless to make an inverse index on an entity-typed one-to-one function. Making an inverse index stores "back pointers" from the referenced entities to the referencing entity. For a value-typed one-to-one function, an index is useful for looking up entities by value, for example, finding a person with a given social security number.

Estimating the Size of an Index

Sometimes Statice can do a slightly better job if you tell it in advance how large to expect the indexed sets to be. That is, if you're making an index to speed up a query, how many results is the query likely to have? In some applications, the designer knows in advance the approximate properties of these sets. So while different departments in a university might have different sizes, one might estimate that departments generally have around 1000 students in them. If Statice knows this, it can do some allocation in advance, and save time and/or space later when the database is filled in.

To provide such a guess, use the **:index-average-size** and **:inverse-index-average-size** options in the function clause. The values should be forms that evaluate to integers.

It is not necessary to pinpoint the exact values of the numbers; just pick a rough ballpark figure. Even if you do not provide any numbers, the added costs are quite small, since Statice dynamically adjusts formats to represent the index in an efficient way.

Multiple Indexes: Indexes on Several Functions

You can create a single index on more than one function of a type. The example shows the use of the **:multiple-index** option to **statice:define-entity-type** to create an index on **instructor-rank** and **instructor-salary**. The functions listed in the **:multiple-index** clause are specified by attribute name, not by the function name.

```
(define-entity-type instructor (person)
  ((rank :string)
   (dept department)
   (salary :integer))
  (:multiple-index (rank salary)))
```

The function **show-well-paid-associate-instructors** shows a query that can be resolved directly from this index.

```
(defun show-well-paid-associate-instructors (salary)
  (with-database (db *university-pathname*)
    (princ
      (with-transaction ()
        (with-output-to-string (string-stream)
          (for-each (i instructor
                       (:where (:and (string= "a" (instructor-rank i))
                                     (> (instructor-salary i) salary))))
            (format string-stream "~%~A" (person-name i)))))))
  nil)
```

Rules for Multiple Indexes

You can create a multiple index when all of the following requirements are met:

- All functions listed are single-valued functions.
- All functions listed are in the same area as the type.
- None of the functions are inverse functions.
- At least two functions are listed.

You can provide many **:multiple-index** clauses.

Guidelines for Multiple Indexes

A multiple index can speed up queries in which there are several clauses combined with **:and** in the **:where** clause, and there are clauses that mention all of the functions in the index, or a leading sequence of them. If the leading sequence is not as long as the whole set, then the index is helpful only if the tests are all equality tests. If the sequence is as long as the whole set, then the last test can be an inequality or string-prefix test as well.

Enforcing Uniqueness on a Combination of Attributes

A **:multiple-index** clause can also specify **:unique t**. This makes Statice enforce a constraint: no two entities can have the same combination of values for all of the functions mentioned in the clause. That is, if you

specify **:unique t**, then a particular set of values for the set of functions mentioned in the clause specifies no more than one entity. This example allows two instructors to have the same rank or the same salary, but makes in impossible for them to have the same values for both rank and salary.

```
(define-entity-type instructor (person)
  ((rank :string)
   (dept department)
   (salary :integer))
  (:multiple-index (rank salary) :unique t))
```

When a multiple index is specified as **:unique t**, it is necessary to initialize the value of at least one of the attributes when making entities. In this case, you can create one instructor without entering a value for the rank or salary attribute; the default for both would be **nil**. However, if you then try to create a second instructor without entering a value for the rank or salary attribute, the default for both would again be **nil**. Since this would violate the uniqueness constraint, Statice would signal an error.

In addition to specifying multiple indexes in the **statice:define-entity-type** forms, you can use the functions **make-multiple-index**, **delete-multiple-index**, and **multiple-index-exists**.

4.3. Statice Type Sets, Attribute Sets, and Areas

At this point it is useful to describe how **statice:for-each** finds entities when no index exists. We'll examine both of the queries in the function **show-students-and-their-courses**. (This example was used earlier in the documentation, and is repeated here.)

```
(defun show-students-and-their-courses ()
  (with-database (db *university-pathname*)
    (princ
      (with-transaction ()
        (with-output-to-string (string-stream)
          (for-each (s student)
            (format string-stream "~%~A:" (person-name s))
            (for-each (c (student-courses s))
              (format string-stream " ~A" (course-title c)))))))))
```

Type Sets

Consider the outer **statice:for-each**, which finds all of the students, and assume there are no helpful indexes. **statice:for-each** finds the students by using an auxiliary structure called a *type set*. A type set is like a B-tree index, but with no keys: it simply points to every entity record of a given type. It lets **statice:for-each** find all the student records, without having to examine all the other records in the

database that are not student records.

Attribute Sets

Now consider the inner **statice:for-each**, which finds all the courses being taken by one particular student. Since **student-courses** is a many-to-many attribute, its value is represented by relationship records. Statice uses an auxiliary structure called an *attribute set* to find all of the relationship records that make up the function **student-courses**, for all students and courses. It examines each one and chooses only those that point to the particular student. This is not nearly as fast as using an index. However, it is a lot faster than checking every single record in the entire database, including the ones that are not part of **student-courses** at all.

An attribute set is very similar to a type set. The only difference is that a type set point to every entity record of a type, while an attribute set points to every relationship record of an attribute.

Cost of Type Sets and Attribute Sets

Type sets and attribute sets have some of the same costs as indexes. Insertion and deletion into the sets have costs; the sets take up disk space; the sets themselves are extra information that must be read in from the database; and they can cause concurrency conflicts.

Controlling the Use of Type Sets and Attribute sets

By default, Statice always creates a type set for every type, and an attribute set for every function that resides in relationship records.

In some cases, it makes sense to tell Statice not to create the sets at all. In particular, if you are sure that any query that your application program does will always be resolved by an index, then you can get rid of the sets entirely. See the section "How to Control Type Sets, Attribute Sets, and Areas", page 131. There is an alternative that is often better, which is to control the use of areas. In order to control areas wisely, you need to understand how paging affects performance.

Paging Considerations

Earlier, we stated that a database is made up out of chunks called records. In addition to the records, there are also auxiliary structures: indexes, type sets, and attribute sets.

A database can also be broken into a different kind of unit called a *page*. Every database is a sequence of pages. Every page is the same size. The page size might vary from one Statice file system to the next, because it depends on hardware criteria that can be different from one processor or disk to the next. The division of databases into pages is a physical phenomenon, imposed on the file system by the way hardware works. (Currently, pages are 1152 bytes on Symbolics 36xx processors, and 1280 bytes on Symbolics Ivory-based processors; the processor type of the server is relevant, not of the client.)

Everything in a database resides in some page. Every page in the database has a specific purpose. Some are used to store portions of indexes. Others are used to store pieces of attribute sets, or type sets. Others are used to store records. In this discussion, we'll concentrate on the pages that are used to store records, which are called *data pages*.

Typically, a record is smaller than the size of a page. Most data pages hold many records. However, there is no limit in Statice on the size of a record, and some records are larger than the page size. One reason this can happen is if one of the values stored in the record is a very long string. In such cases, the record is spread over many data pages. The process of storing records within pages happens automatically.

When you access a database, the amount of time spent accessing the disk is often the dominant cost of the access. This is because disks are so much slower than processors. Whenever the processor accesses the disk, it always transfers an entire page, or several entire pages; this is how disks work. So the cost of using the disk is usually measured in terms of the number of pages that have to be accessed.

This has an important implication. If Statice wants to access seven different records, and all of those records happen to be stored on the same page, then the cost will be only one disk access. But if all seven records are on different pages, the cost will be seven disk accesses, which is seven times the cost. If you can arrange things so that the records that you need at the same time tend to be grouped together into fewer pages, you can access them more quickly.

To understand the physical structure of your database, you do not have to actually be aware of individual pages, and which record is on exactly which page. The important thing to know is that some records tend to be close to other records, and this means that there is a good chance many of these nearby records will be on the same page.

When Statice creates a new record, it finds a data page that has enough room to hold the record, and puts it there. If there is no data page with enough room, it creates a new data page for the record. If there is a lot of interleaved deleting and creating, it is hard to predict exactly where a record will end up being created. But if there is a lot of creation without intervening deletion, records will tend to be close to the records that were created around the same time.

You can affect the speed of Statice queries by controlling which records are stored near other records. There are two ways to do this: areas and clustering.

Areas

An *area* is a set of data pages. Every data page is a member of one particular area. Every type has an associated area, and every attribute that is stored in relationship records has an associated area as well. Entity records of a type are always stored in data pages from the type's area, and relationship records work likewise.

By default, there is only one area, and every type and attribute is associated with that area. All of the records are placed into a single set of data pages.

Suppose you request a second area, and associate the type **student** with that area. Now, the entity records for **student** entities are created in the second area, and everything else is created in the first area. The result is that all of the **student** entity records are crowded into the smallest possible number of pages. So if you do a query that examines every **student** record, it will be faster than it otherwise would have been, because fewer pages need to be accessed.

It is important to keep in mind what information is stored in what kind of records. The **student** records hold the name, id-number, and dept of the student, and these can be examined once the record has been accessed. However, in order to access the courses, Statice has to fetch the relevant relationship records. These are stored in the first area, and so more pages have to be accessed from the disk. In order to access the shirts, Statice has to access the relevant **shirt** entity records, which also are not in the new area.

Cost of Using Areas

What is the cost of using areas in this way? Suppose that when we create students, we generally store all the information about one student, then all the information about the next student, and so on. If everything were in one area, the **student** entity record, and the **student-courses** relationship records that go with it, would tend to be close together, possibly on the same page. Now, if we want to access one particular student, and look at all his courses, it might take only one page access. But if there were a separate area for **student** entity records, it would take at least two page accesses, because the **student** entity records would have to be on a different page from the **student-courses** relationship records.

Guidelines for Controlling Areas

Whether to put something in a separate area or not depends on the kind of accessing you expect to be doing, or the kind whose speed you care the most about. Qualitatively, if you are interested in a few things about many entities, separate areas are better, but if you are interested in many things about a few entities, keeping things together is better.

You can also control the area of a particular function. If you specify a particular area for a set-valued function, the relationships records for that function go into the area. If you specify a particular area for a single-valued function (and it is not the same area as the area of the type), then the values of the single-valued function are not stored in the entity record, but rather live in relationship records, and those relationships records go into the specified area.

Area Scans

When **statice:for-each** does not find any relevant index, and there is no attribute set or type set, it performs an *area scan*. In an area scan, Statice accesses every page of the area associated with the type or function, and examines each record in each page to see if it's what the query is looking for.

We discussed an example in which a new area was established for **student** records. Consider a **statice:for-each** query over the **student** type. If there were no

type set for **student**, **statice:for-each** would do an area scan on this area. Although there is no type set, this is quite efficient. First, Statice accesses only those pages that it needs to. It actually accesses slightly fewer pages than a type set scan would access, since there is no need to access the type set itself. Second, the only records it finds are **student** records, since any other record would not be in this area.

When there is only one type of record in an area, there is no need to have a type set, and in fact the type set only slows things down.

If all the different types of records are in the same area, then the type sets usually speed queries up a lot. The only exception would be if nearly every record in the database is of one type, to the point where just about every single page of the database has at least one of these records in it. Then you might be better off without a type set.

If there are several areas, and one area has a few different record types, then whether a type set (or attribute set) is desirable or not depends on the numbers. Generally, if there is one record type that occupies most of the area, then a type set is less likely to be helpful. If there are only a few such records, the type set is more likely to be helpful.

Commercial relational databases very often put every single record type into its own area, and have no type sets or attribute sets. This is a perfectly reasonable strategy. It is particularly useful if you often access the database by searching through many records of the same type, and more rarely access records together of many different types. Statice provides this as one strategy, but also offers others, because different strategies are better for different databases.

Areas and Indexes

In the section "Introduction to Indexes in Statice", we said that the **:index** option is meaningful only for set-valued attributes. There is an exception to this: if the values of a single-valued attribute are stored in a different area from the entity itself, **:index** is meaningful.

When an entity's attribute value is in a different area from the entity itself, the attribute value obviously cannot be stored in the entity record. Instead, it is stored in a relationship record. In order to fetch the value of the attribute, Statice must locate the relationship record. If there is no index, Statice has to search for the relationship record, using an area scan or a type set scan. With an index, Statice can just follow a pointer.

4.4. How to Control Type Sets, Attribute Sets, and Areas

This section shows examples of controlling the presence of absence of type sets and attribute sets, and controlling which records go into which areas.

Specifying an Area for Types

This example redefines the **student** type to shows the use of the **:area** option to **statice:define-entity-type**. This means that there should be a new area, separate from the default area, designated **student**, and all of the entity records for student entities should be placed in that area. The name space of areas is independent of the names of types and attributes, so you can have both a type named **student** and an area named **student**.

```
(define-entity-type student (person)
  ((dept department)
   (courses (set-of course)))
  (:area student))
```

The relationship records of the **student-courses** function will also be placed into the **student** area, because the default area for an attribute's relationship records is the area of its entity type. If this new definition of **student** replaced the original one in the schema definition, then all **student** and **student-courses** records are placed in the **student** area, and the rest of the records are placed in the default area; there are two areas in the database. Type sets and attribute sets are still created for all types and functions, including **student** and **student-courses**.

Specifying an Area for Functions

This example shows the use of the **:area** option for attributes. The **student-courses** relationship records are now placed into a new area, designated **sc**.

```
(define-entity-type student (person)
  ((dept department)
   (courses (set-of course) :area sc))
  (:area student))
```

The :type-set and :attribute-set Options to statice:define-entity-type

This example shows the use of the **:type-set** entity type option and the **:attribute-set** attribute option. This definition is like the first one: **student** and **student-courses** are both in area **student**, and everything else is in the default area. Even though **student** and **student-courses** are not alone in their own areas, we have nevertheless specified that there should be no type set for **student**, and there should be no attribute set for **student-courses**.

```
(define-entity-type student (person)
  ((dept department)
   (courses (set-of course) :attribute-set nil))
  (:area student)
  (:type-set nil))
```

Storing Entities of Several Types in an Area

This example shows new definitions of the types **student** and **course**. Both specify area **sc**. Therefore, entity records for both **student** and **course** entities, as well as relationship records for type **student-courses**, all are placed into area **sc**. This is the reason that areas have names: so you can mention them more than one place in a schema.

```
(define-entity-type student (person)
  ((dept department)
   (courses (set-of course)))
  (:area sc))

(define-entity-type course ()
  ((title :string :unique t)
   (dept department)
   (instructor instructor))
  (:area sc))
```

Summary

- There is always one default area, designated **nil.**
- The area for entity records is specified by the **:area** option for the type, and defaults to **nil**.
- The area for relationship records is specified by the **:area** option in the clause that defines the attribute, and defaults to the area of the type.
- The presence of a type set is controlled by the **:type-set** option to the type, and default to **t**.
- The presence of an attribute set is controlled by the **:attribute-set** option in the clause defining the attribute, and defaults to **t**.
- The **:area** option is not inherited. It must be specified explicitly for each type. Note that specifying **:area nil** explicitly is not necessarily the same thing as omitting the **:area** keyword.

4.5. Clustering Technique for Statice Databases

When you access information in a database, Statice creates a buffer in the virtual memory of your machine in which to store that information. The buffer contains a page worth of data. The speed of accessing any data on that page is greater than accessing a different page which must be fetched from the database. You can increase the performance of your program if you can predict groups of entities that would need to be accessed together, and specify that they should be stored on the same page, or on a set of pages.

The technique called *clustering* enables you to group related entities on a set of pages. There are two ways of making clusters: using *declarative specifications*, and using *imperative specifications*. The declarative form may be used when your program fits into the declarative model. If not, you can use the imperative form.

The Declarative Model

One entity type is considered the *parent* type. Every entity of the parent type is placed in its own cluster. Some other entity types are *child* types. Every entity of a child type lives in the cluster of one entity of the parent type. There is an entity-type option to **statice:define-entity-type** called **:own-cluster**, which identifies a parent type.

Each child entity type has a single-valued entity-typed attribute, whose type is either the parent type or one of the other child types. There is an attribute option to **statice:define-entity-type** called **:cluster**, which identifies the entity type as a child type, and identifies the attribute as the one that designates the immediate parent.

No entity type can be both a parent type and a child type. No entity type can have more than one **:cluster** attribute. No parent type and no child type can have the **:area** entity type option.

To use the declarative model, you must create your entities in "top-down" order, making the parent first, then the children that refer directly to the a parent, and so on. If this limitation is too restrictive, you can use the imperative specifications.

The relationships that join parents to children have to be one-to-one or one-to-many; they cannot be many-to-many. You can deal with many-to-many by using the the imperative specifications.

Declarative Specifications

There are two aspects of the declarative form which are specified as part of an entity definition. The first is the **:own-cluster** option to **statice:define-entity-type**; this option says that every time one of these entities is specified is created, it should be placed in a new cluster. This means that a new page is allocated, and this page is empty except for the entity being created. Also, this page is not a member of any other cluster.

The second aspect is the **:cluster** option to entity-typed attributes. When Statice makes a new entity of an entity type that has a **:cluster** attribute, and the value of the **:cluster** attribute is provided (whether explicitly or via **:initform**), and that value is in a cluster, then the new entity is placed in the same cluster. Otherwise (if the value isn't in any cluster, or the value is not provided) the entity is allocated normally (without clustering).

There can be only one **:cluster** attribute per entity type.

Imperative Specifications

The imperative form gives you greater control over the clustering process. It is based on the concept of the *current cluster*. By default (at top-level) there is no current cluster. If there is a current cluster, and you make a new entity, the entity is created inside the current cluster. This overrides the declarative interface and any area specifications.

To declare a current cluster, you use **statice:with-cluster** as in these examples:

```
(with-cluster (:new)  body)
(with-cluster (:none) body)
(with-cluster (entity-handle) body)
```

The contents of the first subform of **statice:with-cluster** (either **:new**, **:none**, or *entity-handle*) is evaluated at run time, so it may be a variable.

When **:new** is specified, all the entities that are created in the dynamic scope of body are created in a new cluster. That is, the first entity to be created in the **statice:with-cluster** form is placed in its own cluster, and the rest of the entities are placed in that same cluster.

When **:none** is specified, the effect is to turn off clustering.

When *entity-handle* is specified, all the new entities created in the dynamic scope of body are placed in an existing cluster—that of the *entity-handle*. If *entity-handle* is not a clustered entity, then no clustering takes place, and everything is allocated wherever there is free space on non-clustered pages.

statice:with-cluster also causes the *set-valued links* to be placed in the cluster pages if clustering is enabled. These are records linking set-valued attributes to entities, which exist if the set-valued entity-type attribute has an index.

If both declarative and imperative techniques are used, the override rule is: if there is any "current cluster", the imperative laws apply, and if there is no current cluster, the declarative laws apply.

4.6. Concurrency Control in Statice

Concurrency control is the facility in Statice that lets many client processes operate on the same database at the same time, without getting in each other's way. Concurrency control is completely automatic in Statice; we only discuss it because it can affect performance. By understanding something about how concurrency control is implemented in Statice, you might be able to improve the performance of your application.

4.6.1. How Locking Works in Statice

The concurrency control technique used in Statice is called "two-phase locking, with page granularity", in the jargon of database systems. This section does not try to take the place of a textbook on the principles of concurrency control in

database systems, but it explains enough about locking to help you understand how it affects performance of Statice.

Locking

To control concurrent access to a database, Statice maintains a software object called a *lock* on every page of the database. The state of the lock controls which clients are allowed to access the page. The lock can have any of three states:

Unlocked	No client is using the page; the next client who asks for it will get it. The page is said to be *unlocked*.
Locked for reading	One or more clients are allowed to read the contents of the page, but no client is allowed to write the contents of the page. The page is said to be *locked for reading* by these clients. The clients are said to own the page for reading.
Locked for writing	One client is allowed to read or write the contents of the page. No other clients are allowed to access the page at all. The page is said to be *locked for writing* by this client. The client is said to own the page for writing.

In other words, many processes can share a page if they're only reading the page, but only one process can use a page if it's writing the page.

The concept of "page" was described earlier: See the section "Statice Type Sets, Attribute Sets, and Areas", page 127.

When a client process tries to read a page of a database, and it does not already own the page at all, it must attempt to *lock the page* for reading. If the page is unlocked, or locked for read, the client is granted the lock. Otherwise, the client process must wait until the state of the lock changes.

When a client process tries to write a page of a database, and it does not already own the page for writing, it must attempt to *lock the page* for writing. If the page is unlocked, the client is granted the lock. If the page is locked for reading by this client and no other clients, the lock is *upgraded* to being locked for writing by this client process. Otherwise, the client process must wait until the state of the lock changes.

At the beginning of a transaction, the client process doesn't own any locks. As the transaction proceeds and the client reads and writes various pages, it acquires locks. Finally, when the transaction ends (either commits or aborts), it releases all of its locks. (This pattern of acquiring and releasing locks is technically known as *two-phase locking*.)

Example of locking

Here's how locking works in a typical transaction. The function **impose-salary-minimum** finds all instructors whose salary is less than **this-much**, and sets their salaries to **this-much**. Suppose there is an inverse index on the **salary** attribute of **instructor**, so that Statice will use an index scan.

```
(defun impose-salary-minimum (this-much)
  (with-database (db *university-pathname*)
    (with-transaction ()
      (for-each ((i instructor)
                 (:where (< (instructor-salary i) this-much)))
        (setf (instructor-salary i) this-much)))))
```

impose-salary-minimum starts a new transaction, which we'll call Transaction A. As Transaction A starts, it doesn't own any locks. The first thing Statice does is to read the inverse index to find the locations of the entity records for the relevant instructors. During this operation, Transaction A has to acquire locks for reading on the pages of the database that hold the inverse index.

If some Transaction B is already executing, and has acquired a lock for writing on any of these inverse pages, Transaction A must wait for Transaction B to complete. This could only happen if Transaction B were trying to write the index, which would happen if it were changing the salary of some instructor, creating a new instructor, or deleting an instructor.

After the index scan, Transaction A owns some of the pages of the index for reading. If some Transaction C attempts to write one of the pages of the index that Transaction A is interested in, by changing the salary of some instructor, creating a new instructor, or deleting an instructor, it would have to wait until Transaction A completes. Because of the way the inverse index is organized, it's possible that the only pages Transaction C needs to modify are pages that Transaction A hasn't locked; it all depends on the details of the data values and the size of the index.

Next, Transaction A writes new values into the entity records of some **instructor** entities. It must acquire locks for writing on each of the pages that those entities live in. If any other transaction owns any lock on one of those pages, Transaction A must wait for it to finish. Transaction A also must update the inverse index on the **salary** attribute, which means it needs to upgrade some of its locks on the index pages to locks for writing. FInally, Transaction A commits the transaction, releasing all its locks, and any transactions that were waiting for Transaction A can proceed.

4.6.2. Deadlocks

Simple Two-way Deadlocks

Suppose a database has two pages, which we'll call P1 and P2. Two transactions start using the database, and the following sequence of events occurs:

- Transaction A modifies the contents of page P1, thus acquiring a lock on P1 for writing.

- Transaction B modifies the contents of page P2, thus acquiring a lock on P1 for writing.

- Transaction A attempts to modify the contents of P2. It first attempts to lock P2, but Transaction B already owns P2, so Transaction A waits for Transaction B to complete.

- Transaction B attempts to modify the contents of P1. It first attempts to lock P1, but Transaction A already owns P1, so Transaction B waits for Transaction A to complete.

At this point, each transaction is waiting the other one to complete, but neither one can. Such a situation is called a *deadlock*. Since application programs can access pieces of a database in any order they choose, it's possible for a deadlock to occur at any time.

Statice monitors the state of all transactions, and looks for deadlocks. When it sees that a deadlock exists, it picks one transaction, called the *victim*, and causes that transaction to abort and restart. This breaks the deadlock and allows both transactions to finish.

In our example, suppose that Transaction A is chosen as the victim. Statice aborts Transaction A, and starts it again. P1, of course, is restored to its original contents. Transaction A again tries to lock page P1. But this time, Transaction B is queued up ahead of it, and Transaction B is granted the lock on P1 first. Now Transaction B can finish, and Transaction A can run without interference.

This is one of the reasons that transactions can restart at unpredictable times. This property of transactions was discussed earlier: See the section "Coping with Transaction Restarts", page 37.

N-way Deadlocks

It is possible for three or more transactions to become involved in a deadlock. For example, Transaction A owns a write lock on page P1, and is waiting for a write lock on page P2. Transaction B owns a write lock on page P2, and is waiting for a write lock on page P3. Transaction C owns a write lock on page P3, and is waiting for a write lock on page P1.

Three-or-more-way deadlocks are less likely to occur than two-way deadlocks, but they can happen. Statice knows how to detect such deadlocks and resolve them by aborting one of the transactions.

Upgrade Deadlocks

Another way a deadlock can happen is from the lock upgrade operation. Suppose Transaction A and Transaction B both own a read lock on page P1. Transaction A now tries to upgrade the lock to a write lock. This request cannot be satisfied until Transaction B releases the lock, so Transaction A waits. Transaction B now tries to upgrade the same lock to a write lock, and waits until Transaction A releases it. Again, both transactions are stuck.

This kind of deadlock is probably the most likely to occur in practice, because it doesn't depend on having two running transactions on the database that access

pages in a different order. The transactions can be doing exactly the same thing, in the same order, and an unlucky interleaving can cause a deadlock. Again, Statice knows how to detect and resolve such deadlocks.

Livelock

The policy of Statice is that the "youngest" transaction involved in the deadlock, i.e. the one that started most recently, is picked as the victim. The intent is to prevent a situation in which transactions are continually aborted and retried over and over again, never completing; this situation is sometimes called "livelock".

There is no absolute guarantee that this policy will prevent a "livelock", although it is likely to. Statice will keep retrying a transaction up to a specified number of times, which is 100 by default. After that, it assumes that something is wrong and manual intervention is needed, so the transaction aborts one last time, non-restatably, and signals the condition **dbfs:transaction-retry-limit-exceeded**. The maximum number of retries is controlled by the variable **dbfs:*transaction-retry-limit***.

4.6.3. How Locking Affects Performance

The most important effect of locking on the performance of a program using Statice is that when one transaction holds a lock on a page, other programs cannot access that page, and must wait (unless the lock being held is a read lock, and the other programs only want to read the page). The longer a transaction holds a lock, the more time it "freezes out" other transactions, and the greater the chance that another transaction will be slowed down because it needs to wait for a lock.

Similarly, the longer a transaction holds a lock, the more likely deadlocks become. Deadlocks only arise when two transactions interleave in an unfortunate way. The longer transaction A holds a lock, the greater the time window in which transaction B can enter a deadlock with transaction A.

So the most important result is that you should try to keep transactions as short as possible.

5. Operations and Maintenance of Statice Databases

5.1. The Architecture of Statice

5.1.1. Using Statice Locally or Remotely

Statice can be used on a single host, or between many hosts across the network. This section describes how this works, and explains the terminology used for the participants and the roles they play.

A *host* is a computer, a workstation. A *site* is a collection of hosts (and other things, such as users), all at more or less the same physical place. Every host is at one particular site. We assume that every host at a site is connected to a network, so that each host can communicate with every other host. Hosts, sites, and networks are all described by the namespace database. For more information about the namespace database and the things it describes: See the section "Concepts of Symbolics Networks" in *Networks*. See the section "Setting Up and Maintaining the Namespace Database" in *Site Operations*.

A *Statice File System* is a file system that holds Statice databases. Every Statice File System is at one particular site. Every Statice File System resides on one particular host at that site. Each Statice File System is described in the namespace database by a File System namespace object: See the section "How a Statice File System is Described in the Namespace", page 142. See the section "Attributes for Objects of Type "File System"", page 147.

Suppose that at some site there are two hosts named Mars and Venus, and there are two Statice File Systems named Rose and Iris. Rose resides on host Mars, and Iris resides on host Venus. Fig. 1 shows hosts represented as rectangles, and Statice File Systems represented as circles.

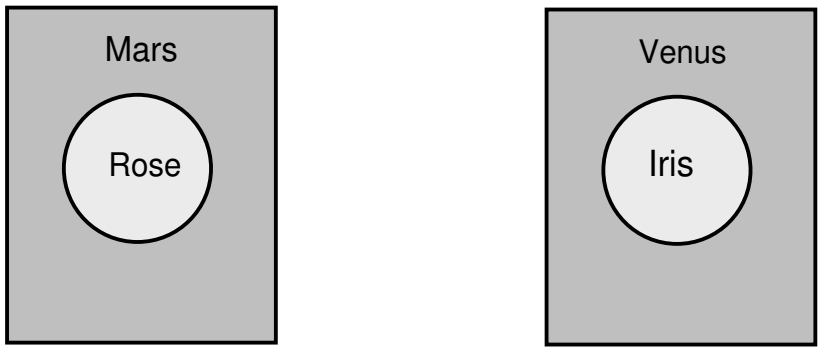

Figure 1. Hosts Mars and Venus, with File Systems Rose and Iris

Now, suppose a process running on host Mars begins using Statice, doing **statice:with-database** and **statice:with-transaction**, calling accessor functions, and so forth. This process might be a Dynamic Lisp Listener, a process associated with

some program defined by **dw:define-program-framework**, or any process at all. A process that calls Statice functions and special forms is a *client* process.

If the client process uses a database that resides in the Statice File System named Rose, Statice notices that Rose is on the same host as the client process itself. We say that the client process is using Statice *locally*, or that the client is accessing a *local database*. When a client is using Statice locally, the client manipulates the database directly, invoking the disk driver to access the host's disks, etc.

If the client process uses a database that resides in the Statice File System named Iris, Statice notices that the Iris is on some other host than the client process. We say that the client process is using Statice *remotely*, or that the client process is accessing a *remote database*. When a client uses Statice remotely, a *server process* is created on the *remote host*, and a network connection is created to allow the client process and the server process to communicate. The client process cannot directly access the disks of another host, and so it delegates this work to the server process.

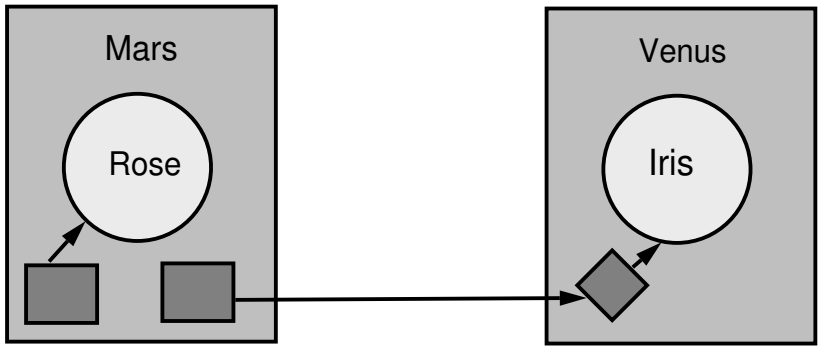

Figure 2. Local and Remote Use of Statice

If a second client process, running on host Venus, accesses the same database on Iris, this second client is using Statice locally. In Fig 2, we have two client processes, both using the same database, one locally and one remotely. In general, there might be any number of client processes accessing a database, many locally and many remotely.

5.1.2. How a Statice File System is Described in the Namespace

Every Statice File System is described by an object in the namespace database. For more information about the namespace database and the things it describes: See the section "Concepts of Symbolics Networks" in *Networks*. See the section "Setting Up and Maintaining the Namespace Database" in *Site Operations*.

The namespace object for a Statice File System is of type File-System (as opposed to Host, Network, User, et. al.), and would typically look similar to this:

```
Host: CHICOPEE
Type: DBFS
Root Directory: FEP1:>Statice>iris.UFD
Pretty Name: IRIS
User Property: PARTITION0 FEP1:>Statice>iris-part0.file.newest
User Property: PARTITION1 FEP1:>Statice>iris-part1.file.newest
User Property: PARTITION2 FEP1:>Statice>iris-part2.file.newest
User Property: LOG-DESCRIPTOR-FILE-ID 1015371-1311996-1048993
User Property: DBFS-DIR-ROOT-FILE-ID 1014172-1311996-1048993
```

When you use the Create Statice File System command, Statice automatically creates a file-system object. See the section "Create Statice File System Command".

The most important attribute is the Host, which says where this file system lives. In this example, Iris lives on host Mars.

The Type field always has the value DBFS. Any other values for this field are reserved for future expansion.

The Root Directory field contains a pathname of a FEP FS file. That file contains the directory of the Statice File System. The pathname should always start with FEP*n*: and end with the .UFD file extension.

The Pretty Name and Short Name mean the same thing as they do in other namespace objects. The Short Name can be used on input as an abbreviation; in particular, you can use it in pathnames. The pretty name is used to display the name of the file system.

The User Properties named PARTITION have values that are pathnames of the FEP files which are the partitions that make up the file system. The database is stored in a number of partitions. For more information on partitions: See the section "Create Statice File System Command", page 167.

The User Property named LOG-DESCRIPTOR-FILE-ID is the unique ID of Statice's "log descriptor" file, an internal file used to store various per-file-system information.

The User Property named DBFS-DIR-ROOT-FILE-ID contains, in the form of a string, the internal unique ID of the special database in the file system that stores the hierarchical directory structure of the file system. This is established when the Statice File System is created, and you should never change it.

For reference documentation on the File-System object and its attributes: See the section "Attributes for Objects of Type "File System"", page 147.

Why is there a separate File-System namespace object? Why doesn't Statice use the host object, as LMFS does? There are three reasons:

- Pathnames starting with "MARS:" already mean "the LMFS found on host MARS". They can't also mean a Statice File System. We must have a different name in order to specify the Statice File System, because the name "MARS" has already been taken. This is the most compelling reason.

- It's possible to have more than one Statice File System on the same host, each with its own name, using different areas of the host's disks. Such a configuration might be desirable for various administrative reasons.

- You can move a Statice File System from one host to another, using removable disk packs, magnetic tape copies, or moving disks. If you do this, you need to change the file-system namespace object for the Statice File System to refer to the new host. All software that refers to the Statice File System by name continues to work, because the file-system namespace object provides the link to the new location of the Statice File System.

5.1.3. Statice Database Pathnames

Many Statice commands take a pathname as an argument, indicating a database. This should be a *database pathname*. The most striking difference between a database pathname and an ordinary pathname is the first component: for a database pathname, this is a Statice File System; for an ordinary pathname, this is a host.

Every Statice File System contains a set of files. Each file contains a Statice database, and is named by a database pathname.

Database pathnames resemble LMFS pathnames. The "host name" part is the name of the Statice File System, followed by a colon. After that are the names of the directories, separated by greater-than characters. The last part of the pathname is the name of the file. For example, the pathname **Iris:>george>financial>ledger** names a file in the Statice File System named Iris. The root directory on Iris contains a directory named **george**, which in turn contains a directory named **financial**, which in turn contains a file named **ledger**.

There are some important differences between database pathnames and LMFS pathnames. Database pathnames have a name, but no type or version. Statice File System directories store fewer properties than LMFS directories; there is no modification date, reference date, do-not-reap flag, and so on. There are also no links, only files and directories. Statice File System directories support the following properties, which can be examined with directory listings (see **fs:directory-list** or **fs:file-properties**).

:author The user ID of the creator of the file.

:creation-date The date and time at which the file was created, expressed as a universal time.

:comment An arbitrary string that appears in directory listings.

:directory **t** if this is a directory, **nil** if this is a file.

:length-in-blocks The length of the file, in blocks. A block is 1152 (decimal) bytes, the size of a FEP file system block. (FEP blocks are 1152 bytes on 3600-family machines, and 1280 bytes on Ivory machines.)

The **:author**, **:creation-date**, and **:comment** properties can be set using **fs:change-file-properties**. **:comment** can be set for files but not directories. The other properties cannot be set.

Database pathnames are case-insensitive for lookup, like those of LMFS. When you make a new file, the Statice File System directory remembers the case you used and stores this in the directory. To look up a file that already exists, you can use either case for any character. A directory cannot have one file named **Foo** and another named **foo**; these are considered to be the same name.

Database pathnames support relative pathname syntax, wildcard syntax, and completion, just like LMFS pathnames. **<foo>bar** means the file named bar in the directory named foo in the directory that is the parent of the default pathname's directory. **>foo>*** means all the files in the directory named foo. **>foo>**>*** means all the files in the directory named foo and its descendants. **>foo>b*** means all the files whose names start with **b** in the directory named **foo**.

There is no undeletion, and no expunging. When you delete a file, it is immediately and permanently removed.

Files cannot be opened by the Lisp **open** function because they are not really files in the sense of the Lisp stream system. The contents of files are accessed only via Statice operations, never by streams. The only exception is if they are opened with a **:direction** of **:probe**, **:probe-directory**, or **:probe-link**. This lets programs use **probe-file** to check for the existence of files in a Statice File System.

Database pathnames work correctly with the Genera directory manipulation tools, such as DIRED, FSEdit (the File System Editor), and commands such as Show Directory, Create Directory, Delete File, and Rename File. However, they do not work with commands that attempt to open files, such as Copy File and Show File.

The root directory of every Statice File System directory contains an entry named **Directory**, which refers to the directory itself. This gives you a way to name the root directory as a file, in order to perform operations on it. This is rarely necessary, and you can usually just ignore the **Directory** entry.

5.1.4. Dealing with Databases by Their Pathnames

Many operations on databases can be done by using normal file commands on the database pathname. See the section "Statice Database Pathnames", page 144.

- What databases are stored in a directory of a given Statice file system? Use the Show Directory command:

```
Show Directory beet:>fred>*
```

- What are all the databases stored an entire Statice file system? Use the Show Directory command:

```
Show Directory beet:>**>*
```

- How can I rename a database? Use the Rename File command.

```
Rename File beet:>university beet:>harvard
```

- How can I remove a database? Use the Delete File command on a database pathname.

  ```
  Delete File beet:>university
  ```

  Although this method permanently removes the file, you can restore the file from backup tapes (if you have any). See the section "Selective Restore Command", page 161.

5.1.5. Services and Protocols Used by Statice

Statice uses several network services and protocols. The commands that install Statice at your site add new information to your namespace host objects, indicating that these services and protocols are supported. In this section, we briefly describe the new namespace information; this information is not required in order to write Statice programs, but it might be useful to help you debug any unusual namespace-related problems.

DBFS-PAGE Service

DBFS-PAGE is the network service provided by a Statice server for the benefit of Statice clients. When a Statice client first accesses a particular Statice server host, it invokes the DBFS-PAGE service on that host. From then on, this client uses this connection to communicate with that host, even if it accesses more than one Statice File System on that host.

When the transaction ends, the connection is returned to a free pool, so subsequent transactions can use that connection. This helps reduce the amount of time spent opening network connections.

Statice servers use the same connection scavenger mechanism used by Genera file servers, so that if a connection hasn't been used for a long time, the server process is killed and the connection goes away. See the section "The File Control Lifetime Host Attribute" in *Site Operations*.

The namespace entries for the DBFS-PAGE service should be present in the host object of every host used as a Statice File System server. The Add DBFS Page Service command sets up this service entry in the namespace database. See the section "Add DBFS PAGE Service Command", page 165.

See the section "Using Statice for the First Time", page 17.

DBFS-Page Protocol

DBFS-PAGE is the name of a network protocol that implements the DBFS-PAGE service. This protocol is built on top of the byte-stream-with-mark network medium. See the section "BYTE-STREAM-WITH-MARK Network Medium" in *Networks*. It can be used with Chaosnet or TCP/IP networks. For Chaosnet, the contact name is "DBFS-PAGE"; for TCP/IP, the port number is 569.

ASYNCH-DBFS-PAGE Service

ASYNCH-DBFS-PAGE works in the reverse direction: the Statice server uses this service to form a connection back to Statice File System client hosts. The server uses this connection to notify the user when pages have been modified by another client; the client acts on this information by invalidating its cache.

The service that ASYNCH-DBFS-PAGE provides is not necessary for correct operation of Statice. If the server finds that it cannot form a connection to the client, it simply gives up and tries again later. The cache coherency protocol within Statice makes sure that invalid data is never used. However, Statice will be more efficient if ASYNCH-DBFS-PAGE is working properly. You should try to make sure that all hosts that use Statice as a client have the proper namespace entries for ASYNCH-DBFS-PAGE in the namespace database. The command Add ASYNCH DBFS PAGE Service should be run on each client host: See the section "Using Statice for the First Time", page 17.

ASYNCH-DBFS-PAGE Protocol

ASYNCH-DBFS-PAGE is the name of the network protocol that implements the ASYNCH-DBFS-PAGE service. This protocol is built on top of the byte-stream-with-mark network medium. See the section "BYTE-STREAM-WITH-MARK Network Medium" in *Networks*. It can be used with Chaosnet or TCP/IP networks. For Chaosnet, the contact name is "ASYNCH-DBFS-PAGE"; for TCP/IP, the port number is 568.

5.1.6. Attributes for Objects of Type "File System"

Host

Specifies the host that the file system resides on; a host object (required).

```
Host: MARS
```

Type

Must always be DBFS (required). Other values are reserved for future expansion.

```
Type: DBFS
```

Root Directory

Specifies a pathname of a FEPFS file. That file contains the directory of the Statice File System. The pathname should always start with FEP*n*: and end with the .UFD file extension.

```
Root Directory: FEP1:>Iris.UFD
```

Pretty Name

Specifies a name for the file-system to use when showing the name; a token (required).

```
Pretty Name: Iris
```

Nickname

Specifies alternate names for the network; a set of names. The file system may be found by these names.

```
Nickname: IRE
```

Short Name

Specifies additional nicknames; a set of names. A short-name is used when a program wants to display a host's name without using up too much space. A short-name is used for both input and output. This is also used in the printed representation of pathnames.

```
Short Name: I
```

User Property

User-Property

All objects contained within the namespace (hosts, sites, namespaces, printers, and users) are eligible to have a User-Property attribute. It consists of a pair whose first element is an indicator (like that of a property list) and whose second element is a token. The User-Property attribute holds any information that users choose to associate with an object. For example:

```
User-Property: ID-number 123-45-6789
```

Statice automatically places several user properties into file-system objects. The User Properties named PARTITION have values that are pathnames of the FEP files which are the partitions that make up the file system. The database is stored in a number of partitions. For more information on partitions: See the section "Create Statice File System Command", page 167.

The User Property named LOG-DESCRIPTOR-FILE-ID is the unique ID of Statice's "log descriptor" file, an internal file used to store various per-file-system information.

The User Property named DBFS-DIR-ROOT-FILE-ID contains, in the form of a string, the internal unique ID of the special database in the file system that stores the hierarchical directory structure of the file system. This is established when the Statice File System is created, and you should never change it.

5.1.7. FEP File for Generating Statice Unique IDs

When you first run Statice on a machine, it automatically creates a small file (2 blocks) in the FEP file system, whose name is:

```
FEPn:>UNIQUE-ID.FEP.1
```

where *n* is the lowest fixed-medium disk unit, or the lowest disk unit if all units are removable-medium. (In almost all configurations, *n* is zero.)

This file is used internally by Statice to generate unique IDs. We recommend that you leave it there, and don't delete it. If you do delete it, Statice will re-create it next time Statice is run.

5.2. Statice File System Operations Program

The Statice File System Operations program is an interactive utility for maintaining and manipulating Statice File Systems. It serves primarily as the user interface to the backup system. It also provides commands for enabling and shutting down a Statice File System, and other functions.

We first present several sections that give background information, and then describe how to use the program itself, in the section "Using the Statice File System Operations Program".

Some operations on databases can be done by using normal file commands on the database pathname. For details: See the section "Dealing with Databases by Their Pathnames", page 145.

5.2.1. Overview of the Statice Backup Facilities

The Statice File System backup facilities let you make backup copies of the databases stored in a Statice File System, onto another medium. If the disk holding the Statice File System is damaged or destroyed, the copy of the Statice File System can be restored onto a fresh disk.

It is very important to back up your Statice File System on a regular basis. Disks are fragile and subject to failure. Doing backups is the only way to protect your data against disk failures.

The backup facilities copy database information to tertiary storage media. Currently, industry-standard magnetic tape and cartridge tapes are supported. The software names and catalogs the media into groups called volume sets.

Two forms of backup are provided: complete backups, and continuous archive backup. (Currently only complete backup is supported.) Complete backup makes a complete copy of a database file system onto tertiary storage, and assures the copy is transaction-consistent. If the database file system is destroyed, you can restore the copy, losing only changes made since the latest backup copy was produced. Archive logging continuously copies all database changes to tertiary storage. If the database file system is destroyed, you can restore the latest complete backup copy, and then replay the archive changes, so that no information is lost.

A backup tape from a Statice File System stored on a 3600-family Statice server cannot be reloaded into an Ivory Statice File System, and vice versa. Also, if you want to move whole databases between file systems of different block size, you have to use the high-level dumper (the Dump Database and Load Database commands) to convert the data into a text file, and then move the text file.

See the section "High-level Dumper/Loader of Statice Databases", page 163.

5.2.2. Kinds of Tertiary Storage

Backup copies are kept on *tertiary* storage. (Primary storage is main memory, and secondary storage is disk.) The Statice File System backup facilities back up to and restore using a *generic tertiary storage protocol*, so that different kinds of tertiary storage can be used interchangably, and new kinds can be added in the future.

With Symbolics 36xx systems, two kinds of tertiary storage are currently supported: 1/4-inch cartridge tapes, and 1/2-inch industry-standard reel-to-reel magnetic tapes. (The fraction of an inch refers to the width of the tape itself.) Both kinds of tapes can be used either locally or remotely. In local usage, the tape drive hardware is physically connected to the workstation doing the backup or restore. In remote usage, the tape drive hardware is connected to some other computer, which communicates with the workstation over the network.

In future releases, we we plan to support write-once optical disks as another tertiary storage medium. We also intend to support other formats of magnetic tape as hardware support becomes available.

The physical integrity of backup copies is very important. If it's necessary to restore a file system from a backup copy, it is imperative that the data be readable from the copy. Unfortunately, magnetic tapes are an imperfect physical medium. Periodically, tapes are found to be unreadable, or to contain data errors.

To protect against problems with tapes, the backup system always writes data using a powerful error-correcting coding technique, a simple version of Youngquist's algorithm. The technique is effective at recovering from large damage spots on tape; this is important, since it is not uncommon for 100 sequential bits to "drop out". The cost of the algorithm is that approximately 3/2 as much tape is required to store the same amount of information. We believe that the protection is worth the cost of the extra tape. We tested this coding technique by scratching a tape with a razor, and the data were still recovered.

If you have a choice between industry-standard and cartridge tape, industry-standard tape is preferable, because:

- Industry-standard tape runs faster than cartridge tape.
- More data fits on a single industry-standard tape than on a cartridge tape, so you don't have to change tapes as often.

The primary advantage of cartridge tape is its lower cost. Every Symbolics site has

at least one cartridge tape drive, because cartridge tape is used for distributing software.

The benefits of using an industry-standard tape drive increase if you have large Statice File Systems, or if you write new data frequently.

5.2.3. Choosing the Kind of Tertiary Storage to Use

If you have a choice between local and remote usage, local usage is preferable, because:

- Network connections are inherently unreliable. It is inconvenient to have a network connection fail during a backup or restore operation.
- Local usage runs faster than remote usage.

Therefore, we recommend that you put your Statice File System on a workstation that has its own tape drive, if possible.

When a command of the Statice File System Operations program prompts for a "device specification", it wants to know which tape drive to use. (The prompt doesn't say "tape drive" because there will be other kinds of tertiary storage in the future.) The default is to use the cartridge tape drive on the local workstation. You can change the host, to use a tape drive on another computer, and you can change the device type to industry-standard tape.

If you select industry-standard tape, the prompt expands to let you specify a unit number and a density. The unit number parameter is provided because it is possible to attach more than one industry standard tape drive to a computer; the unit number distinguishes which one you want. The default value is zero, and if there is only one tape drive, it should be unit zero.

Industry standard tapes can be written at several different densities, measured in bits per inch. We support 1600, 3200, and 6250 bits per inch, defaulting to 3200. Not every tape drive can read and write at every density! You must check the hardware you intend to use, and determine what densities it supports.

5.2.4. Volume Capacity

When you make a backup copy, you don't need to know in advance how many volumes will be needed to hold the copy. Whenever the dumper reaches the end of one volume, it asks you to mount another volume, until the copy is completed.

However, you might want to be able to estimate, in advance, how many tapes will be needed to hold a copy. Here is some information to help you make such an estimate. (There is no requirement that you make such an estimate, but it is sometimes desirable.)

It's difficult to make an accurate estimate of how many volumes are needed for a backup copy, because the number depends on many factors. More volumes are needed if there are many small files than one large file. Different tape drives have

different characteristics; industry-standard tape drives don't all write interrecord gaps the same way, and cartridge tape drives are affected by the quality of the tape medium and the cleanliness of the heads. So the following numbers should be treated as approximate figures.

The numbers below show the actual data capacity for several different kinds of tape. The actual data capacity is smaller than the raw capacity primarily because of the overhead of the error-correcting coding used in Statice File System backup tapes. Capacities are given in megabytes.

DC300XL/P cartridge tape: 30 MB

DC600A or DC600XTD cartridge tape: 40 MB

600-foot industry standard tape at 3200 bpi: 14 MB

2400-foot industry standard tape at 3200 bpi: 60 MB

Industry-standard tapes can be written at different densities; the numbers above assume a density of 3200 bpi. If you are using 1600 bpi, divide the numbers by two; if you are using 6250 bpi, double the numbers. Similarly, if you use a reel of tape with some other length, apply the appropriate ratio.

The sizes of the databases in a Statice File System are shown in Statice File System directory listings, in blocks. A block is 1152 bytes. So, for example, if you had a Statice File System containing four databases, two 1000 blocks long and two 2000 blocks long, the total amount of data is about 6 MB, which will easily fit on one tape of any kind.

5.2.5. Tertiary Volumes and Volume Sets

A *volume* means one physical tertiary storage medium, such as one reel or one cartridge of tape. (In future releases, a single write-once optical disk will also be referred to as a volume.)

A *volume set* is a set of one or more volumes. Volumes are grouped into sets because each volume is of a fixed size, whereas you can keep expanding the size of a volume set by adding more volumes.

Each complete dump of a database file system is put on its own volume set. The number of volumes in the volume set depends on the size of the database file system. If the Statice File System is not very large, a complete dump fits on a single volume, which constitutes a single volume set.

Each volume set has a *name*, and each volume within the volume set has a *sequence number*. The name is a character string. No two volume sets can have the same name. Names are compared ignoring case, so you must not have one volume named "Foo" and another named "foo". Volume set names may not use character styles or non-standard character sets. The first volume of a volume set should be numbered 1, and each sequential volume is assigned the next number.

When you're using the backup system and a tape is being mounted, you are prompted for a "mount specification". This consists of a volume set name, a se-

quence number, and a device specification. It means that you are mounting a particular volume on a particular device. To specify the device, you are prompted for a host name, a device type, and further parameters depending on the type of the device. For details of device specs: See the section "Choosing the Kind of Tertiary Storage to Use", page 151.

5.2.6. Labels on Volumes

When a volume is being used by the Statice File System backup system, some special identification information is written at the front of the volume, called the *label*. The label includes the volume set name and the sequence number, which uniquely identify the volume. The backup software uses the label to help assure that you have mounted the tape you intended to mount, in order to guard against mistakes.

When you first use a brand-new tape for Statice File System backup, you should write a label on it, by using the Initialize Backup Volume command in the Statice File System Operations program. Be careful to only use this command on fresh, unused tapes, because it will destroy any information already on the tape.

When the backup system is making a backup copy, and it is ready to write onto a new tape, it first reads the label of the tape, to make sure that this tape is the right one. For example, suppose you are making a backup copy to a volume set named FULL0013, and the backup system just finished writing volume number 2 of the volume. It prompts you with the message:

```
End of volume FULL0013/2.  Enter mount specifications for the next volume:
```

You enter a mount specifiction, in which you enter a volume set name and sequence number. The default is volume set FULL0013, sequence number 3. You also physically mount the correponding tape on the device that you specify in the mount specifiction. After you click on Done or press End, the backup system reads the volume label, to make sure that this volume is really volume number 3 of volume set FULL0013. If the label is present and contains what backup expects, the backup copy continues. Otherwise, you are given several options:

Accept the information on the volume
: (*You believe you entered the wrong thing, and you want to use what the tape says.*) This choice tells the backup system to use the volume set name and sequence number that were found in the label on the tape, even though they aren't what we originally expected. This sets the backup system's concept of "current volume", so the backup system will expect the next volume to follow this one.

Remount a different volume
: (*You believe you mounted the wrong tape, and wish to try mounting a different one.*) This choice lets you remove this volume from the drive and try another one. The backup system prompts you with "Is the desired volume mounted?" When you answer "y", it proceeds to read the label of the new tape and check it.

Reenter different volume specifications
: (*You believe you entered the wrong thing, and you want to try to enter it again.*) This choice makes the backup system return to the point where you were prompted for mount specifications for this tape; you are prompted again. Use this if the reason for the problem is that you didn't provide the right mount specification.

Overwrite the volume with the specified information
: (*You believe that what the tape says is wrong, and you want to use what you entered.*) This choice tells the backup system to ignore what the label says, and write the tape using the mount specification that you provided. This overwrites the label and can change the volume set name and sequence number in the label.

Abort the current operation
: (*You don't want to proceed further.*) This choice returns you to the Statice File System Operations program.

If the tape you mount is a fresh, unused tape, you'll get this list of choices, and you can select [Overwrite] to write a new label on the tape. So it is not actually necessary to use the Initialize Backup Volume command to write an initial label; [Overwrite] also does it for you. However, there's a drawback to relying on [Overwrite]. When the backup system tries to read the label of the blank tape, the tape drive attempts to read data from the tape, but cannot find any. Some drives react to this lack of data by reading further and further down the tape, hoping to find data eventually, which can take a long time. So if you don't run Initialize Backup Volume on new tapes, backup copying might be substantially slower.

This behavior is part of the tape drive and tape controller, and cannot be modified by software. Currently, all industry-standard tape drives sold by Symbolics exhibit this behavior, but the cartridge tape drives do not. Therefore, it is particularly time-saving to use Initialize Backup Volume on industry-standard tapes.

5.2.7. Volume Libraries

When you do Statice File System operations, Statice maintains a database called the volume library. This database is stored within the file system. The volume library stores the following information:

For every backup volume that holds information from this file system:

- Volume name and sequence number
- Completion date, which is the date and time this tape was last written
- Type, which is either "Industry Standard Tape" or "Cartridge Tape"
- The tape spec that was used when the tape was last written, which includes the host, and also unit and density for industry-standard tapes

For every backup run (that is, for every time that a dump as made):

- Completion date, which is the date and time this run was performed
- The set of volumes that were written.
- Whether the run is "valid", or whether something went wrong during the dump

For every file that has been dumped:

- Name of the file
- The set of backup-notes (see below)

For every copy of a file that exists on tape, a "backup note", consisting of:

- The file attributes (length, creation-date, author, comment)
- Which volume the copy resides on
- Which backup run this copy was part of

5.2.8. Using the Statice File System Operations Program

Before using the Statice File System Operations program, you must load it by entering:

```
Load System DBFS-Utilities
```

You enter the Statice File System Operations program by entering:

```
SELECT symbol-sh-D
```

The commands appear a menu in the top pane. You can click on them, or type the names of the commands. Typically, an AVV menu will be displayed, prompting you for several fields. When you finish entering the information and press `END`, the command is executed.

We summarize the commands here. For complete documentation on each command: See the section "Dictionary of Statice File System Operations Commands", page 157.

Backing Up and Restoring a File System

The most common operations are Complete Backup and Complete Restore. Selective Restore can also be used, to restore one or more databases from tape to the file system.

Complete Backup Command
: Copies a Statice File System (and all databases in it) to tape.

Complete Restore Command
: Copies a file system (and all databases in it) from a tape to a Statice File System.

Selective Restore Command
: Copies selected databases from a tape to a Statice File System.

Initialize Backup Volume Command
: Writes the volume number on the tape label itself.

Getting Information

Describe Statice File System Command
: Displays information about a file system.

Describe Backup Volume Command
: Displays information about a backup volume stored on a tape.

Show Backup History Command
: Displays information about all backup runs done on a file system.

Compare Backup Volume Set Command
: Compares a file system stored on tape with a file system stored on the local machine.

Enabling and Disabling a File System or All of Statice

Enable Statice File System Command
: Enables a file system: allows transactions to be started.

Disable Statice File System Command
: Disables a file system: aborts any transactions in progress and disallows any transactions to occur until the file system is enabled again.

Enable Statice Command
: Re-enables Statice activities.

Disable Statice Command
: Entirely disables all Statice activities.

File System Manipulation

Create Statice File System Command
: Creates a new Statice File System on the local host.

Delete Statice File System Command
: Expunges an entire Statice file system, and removes all traces of it, including every database in it; this is a very dangerous command.

Show All Statice File Systems Command
: Lists all the file system objects and the host that they reside on in a namespace.

Add Statice Partition Command
: Adds a new partition to an existing Statice file system.

Show Statice Partitions Command
: Lists the partitions of a Statice file system, and shows the amount of free space remaining in each partition.

5.2.9. Dictionary of Statice File System Operations Commands

This section documents the commands that are available only in the Statice File System Operations program.

The following commands are available from the Statice File System Operations program, but are described elsewhere because they are also available at top level.

- "Add Statice Partition Command"
- "Create Statice File System Command"
- "Delete Statice File System Command"
- "Show All Statice File Systems Command"
- "Show Statice Partitions Command"

For documentation on the other Statice commands that can be used at top level: See the section "Dictionary of Statice Commands", page 165.

Complete Backup Command

Complete Backup

Copies all databases in a Statice File System to tape. This command is available in the Statice File System Operations Program.

The AVV menu looks like this:

```
Enter complete backup parameters (volume name required):

File system to backup: DLW-UFS-3
Volume set name: abc
Volume sequence number: 1
Device: Industry-Standard-Tape  Cartridge-Tape
Volume host: ALDERAAN
Show Detailed Progress: Yes  No
```

The file system to backup must be stored on the local host. We discuss volume set names and sequence numbers elsewhere: See the section "Labels on Volumes", page 153.

The volume host is the host that will write the data to tape; it need not be the local host. If that host does not have TAPE service in its namespace object, you will get this error:

```
Error:  Host does not support TAPE service.
```

You can use one of the proceed options to try TAPE service on various mediums such as TCP or CHAOS. You can also edit the host's namespace object to add that service entry, so the error won't happen again.

Symbolics recommends RTAPE via TCP rather than RTAPE via ChAOS (TCP is more reliable). Add the service `TAPE TCP RTAPE` to the host's namespace object.

If you are writing to a blank tape, you will get an error stating that the volume set name does not match that of the tape itself. This is to be expected (because the tape doesn't have a volume set name at all yet). One way to prevent this error is to use the Initialize Backup Volume command before doing Complete Backup, to write the volume set name on the tape. In any case, one of the choices is to Overwrite the tape, which is the correct choice for a blank tape.

Compare Backup Volume Set Command

Compare Backup Volume Set

Compares a file system stored on tape with a file system stored on the local machine. This command is available in the Statice File System Operations Program.

The AVV menu looks like this:

```
Enter specifications for compare:

File system to compare: DLW-UFS-3
Volume set name: abc
Volume sequence number: 1
Device: Industry-Standard-Tape  Cartridge-Tape
Volume host: ALDERAAN
Show Detailed Progress: Yes  No
```

If anyone has made changes to a database in the file system since the backup was done, Statice will report how many pages are different. The output looks like this:

```
File SCRC|DLW-UFS-3:>Volume-Library>%volume-library has 21 pages
different from the tertiary image.
These differences could be caused by updates which occured
between the time the backup volume finished writing and the time
the comparison finished.
```

Complete Restore Command

Complete Restore

Copies all databases from a tape to a Statice File System. This command is available in the Statice File System Operations Program.

The AVV menu looks like this:

```
Enter specifications for restore:
```

```
File system to restore: DLW-UFS-3
Volume set name: abc
Volume sequence number: 1
Device: Industry-Standard-Tape  Cartridge-Tape
Volume host: ALDERAAN
Show Detailed Progress: Yes  No
```

The choices are the same as for the Complete Backup command: See the section "Complete Backup Command", page 157.

The first thing that a Complete Restore does is delete all databases in the file system. It then copies the databases from tape to the file system. Note that if you are writing to a file system that already exists, you will get this message:

```
Before a file system can be restored, the existing files must be
destroyed.  All the files in file system SCRC|DLW-UFS-3 are
about to be destroyed.  Are you sure you want to continue? (Yes
or No)
```

If you want to keep the existing file system, choose No. If you want to restore the file system from tape, choose Yes.

Describe Backup Volume Command

Describe Backup Volume

Displays information about a backup volume stored on a tape. This command is available in the Statice File System Operations Program.

This command gets its information from the tape itself.

The AVV menu requests the type of tape (whether cartridge or industry standard), and the volume host (the host where the tape is mounted).

Describe Statice File System Command

Describe Statice File System *file-system-name*

Displays information about a file system. This command is available in the Statice File System Operations Program.

This command gets information from the file-system namespace object. If the file system is local, it also displays information about how much data is stored in each partition.

Disable Statice Command

Disable Statice

Entirely disables all Statice activities:

- Every currently-running transaction is aborted, signalling the error **dbfs:system-shutdown-transaction-abort**, whose error message is "Transaction aborted due to a system shutdown."
- All server processes executing on the local host are killed.
- All local file systems are shut down. See the section "Disable Statice File System Command", page 160.
- The local host is marked as disabled, so that any new file systems created on the host are automatically disabled.
- All network connections associated with Statice services are killed.

Any attempts to start a transaction while Statice is disabled signal an error. Any attempts by a remote host to connect to this host are rejected. This command is available in the Statice File System Operations Program.

Disable Statice File System Command

Disable Statice File System *file-system-name*

Disables a file system: aborts any transaction in progress that owns a lock on any page of any file in the file system. Disallows any transactions to occur until the file system is enabled again. If any transaction attempts to use a database in the file system, the error **dbfs:file-system-disabled** is signalled. This command is useful if you want to perform administrative activities on a database, and want to make sure nobody else accesses the database. Note that it is not necessary to disable a Statice file system to do a backup dump.

file-system-name must be the name of a Statice file system on the local host. This command is available in the Statice File System Operations Program.

Enable Statice Command

Enable Statice

Re-enables Statice activities. Undoes the effect of Disable Statice, returning Statice to normal. This command is available in the Statice File System Operations Program.

Enable Statice File System Command

Enable Statice File System *file-system-name*

Enables a file system: allows transactions to be started. Undoes the effects of Disable Statice File System, returning the file system to normal. This command is available in the Statice File System Operations Program.

Initialize Backup Volume Command

Initialize Backup Volume

Writes the volume number on the tape label itself. This command is available in the Statice File System Operations Program.

This command is not a necessary step, because Complete Backup will also write the information on the tape, but it prevents the mismatch that can occur when writing a blank tape: the volume number you specify won't match that of the tape (because a blank tape won't have any volume number on it).

Selective Restore Command

Selective Restore

Copies selected databases from a tape to a Statice File System. This command is available in the Statice File System Operations Program.

The AVV menu looks like this:

```
Enter specifications for selective restore:

File system: DLW-UFS-3
Disable file system: Yes  No
Paths to restore: >test2, >test7, and >test8
Repatriate action: Yes  No  Query
Name conflict resolution action: Leave
                                 Rename Existing File
                                 Replace Existing File
                                 Load Into Unique File
                                 Query
Volume selection mode: Automatic  Manual
Device: Industry-Standard-Tape  Cartridge-Tape
Volume host: ALDERAAN
Show Detailed Progress: Yes  No
```

If you choose Yes for "Disable file system", Statice does a Disable Statice File System before doing the Selective Restore, and an Enable Statice File System afterwards. This option is useful if you are performing some kind of delicate operation on the file system, such as recovering from a problem, and you want to make sure that no other users make any changes in the file system while the Selective Restore is in progress. In the general case, it is not necessary to do this.

The "Paths to restore" are pathnames of databases within the file system. They should start with the greater-than sign, >. They are separated by commas. These pathnames can contain wildcards.

Once you press `END`, Statice searches the volume libary to figure out which volume the file is on. You will then be prompted to mount that volume:

```
Is volume abc/1 mounted for restoring? (Y or N)
```

When Selective Restore is searching the volume library to figure out which volume to retrieve a file from, there might be more than one volume that has this file on it. In such a case, it chooses the volume with the most recent completion date (the date and time at which the dump was completed); that is, it uses the most recent backed-up copy.

The repatriation action should never be needed by applications programmers, so you should use the default (no) for this choice.

Show Backup History Command

Show Backup History *file-system-name*

Displays information about all backup runs done on a file system. This command is available in the Statice File System Operations Program.

This command gets its information from the volume library:

Volume Libraries

When you do Statice File System operations, Statice maintains a database called the volume library. This database is stored within the file system. The volume library stores the following information:

For every backup volume that holds information from this file system:

- Volume name and sequence number
- Completion date, which is the date and time this tape was last written
- Type, which is either "Industry Standard Tape" or "Cartridge Tape"
- The tape spec that was used when the tape was last written, which includes the host, and also unit and density for industry-standard tapes

For every backup run (that is, for every time that a dump as made):

- Completion date, which is the date and time this run was performed
- The set of volumes that were written.
- Whether the run is "valid", or whether something went wrong during the dump

For every file that has been dumped:

- Name of the file
- The set of backup-notes (see below)

For every copy of a file that exists on tape, a "backup note", consisting of:

- The file attributes (length, creation-date, author, comment)
- Which volume the copy resides on
- Which backup run this copy was part of

This command first iterates over all the backup runs, in order of completion date (most recent first). For each run, it tells you whether the run is valid, and what

the completion date is. Then it iterates over all the volumes in that backup run, sorted by completion date. For each volume, it prints out the name/sequence-number, the type, the completion date, the host, the unit (if any), and the density (if any).

5.3. High-level Dumper/Loader of Statice Databases

The Dump Database and Load Database commands invoke a "high-level" database dump/load tool. High level means that it dumps and restores the data in a "source" format, rather than in a binary page format as does the dump/restore tool available from the Statice Operations Menu. The high-level dump/load tool can used for certain types of database reorganization operations.

Dump Database always dumps the databse in a transaction-consistent state, because it uses one long transaction to do its job.

The high-level dumper/loader is useful for several purposes. It enables you to:

- Move a database from one place to another, over a channel that can handle only ordinary text.
- Store the contents of a database on some kind of storage medium (e.g. a particular tape format) that can handle only ordinary text.
- Edit the text file to reorganize the database.

Limitations of the High-level Dumper/Loader

- It does not dump to tape, so the size of the dump is limited to the amount of available file server disk space. Further, since the format is "source" level, it may actually take more disk space to dump a database using the Dump Database command than it does to store it.
- You shouldn't do dumping while other users are operating on the database, since it dumps everything in one large transaction (which will lock them out, or else cause the dump/restore facility to abort a transaction). Loading data back into a database can be rather slow since it does many small transactions (to avoid growing the log).
- Because of LMFS file size limitations, it may not be appropriate to dump a database to a LMFS file. This size limitation is approximately 15MB. Instead, you may have to dump to a UNIX or VMS machine with enough disk space.

Clustering is Maintained

Clustering is maintained across dumps. The actual entities may be grouped differently within the clusters, but they will all be in the same cluster that they were before.

Format of the Dump File

The dump file consists of lists which contain information about the contents of the database. At the beginning of the file is information about the real schema, and following that is information about the actual contents of the database. The format of each list is that the first element of it is a keyword symbol specifying what the list is about and the rest of the list is information specific to that type of list.

Schema information is contained in lists which begin with the :DOMAIN, :RELATION, :COMMIT-DOMAINS, and :INDEX keywords. For the most part, users should not modify these lines unless it is obvious what they mean. For example, consider the following definition, which is taken from the file SYS:STATICE;EXAMPLES;BOOKS.LISP:

```
(define-entity-type account ()
  ((name string :unique t :cached t :inverse account-named)
   (balance single-float :cached t)
   (type (member checking owner) :cached t)))
```

The corresponding information in the dump file looks like this:

```
(:RELATION "ACCOUNT" 1 NIL (("%$OF" "ACCOUNT" T T NIL NIL NIL) ("TYPE"
 (CL:MEMBER BOOKS:CHECKING BOOKS:OWNER) NIL NIL NIL NIL NIL) ("BALANCE"
 CL:SINGLE-FLOAT NIL NIL NIL NIL NIL) ("NAME" STRING T NIL NIL NIL NIL)))
```

This information appears on one line in the dump file. If we wanted to change the balance attribute's data type from single-float to double-float, then we'd edit the "CL:SINGLE-FLOAT" piece of text above. Of course if you change the data type of an element like this, you'd also have to change all the data in the file, too.

After the schema information, the actual data in the file is dumped to the file using :ENTITY and :RELATION-DATA entries. The format of an :ENTITY entry is:

```
(:ENTITY ("ENTRY" 14352 3388287 7463818 3147232 NIL))
```

This is a list of an entity's type name (ENTRY), its internal record ID (14352), its unique ID (three 32-bit fixnums), and its cluster ID. For the most part, these should be of little interest to the users.

Users are more likely to be interested in the :RELATION-DATA entries. An account entity might look like this:

```
(:RELATION-DATA "ACCOUNT" NIL (17441 BOOKS:OWNER 200.53 "Lane"))
```

To change the value of the balance for the "Lane" account, you'd change the value 200.53 to another value. You can find out the order of the attributes of this by looking at the real-schema data at the beginning of the file. The attribute order information will be embodied in the :RELATION entry. For example, the :RELATION line above shows that the order of the attributes in the above :RELATION-DATA entry are %$OF (an internal attribute), the type attribute, the balance attribute, and the name attribute.

For more detailed information, you can read the comments in the code in the file SYS:STATICE;UTILITIES;MODEL-DUMPER.LISP.

6. Dictionary of Statice Commands

This section documents the commands that are available in the command processor. Statice also offers a set of commands that are available only in the Statice file System Operations menu. For documentation on those commands: See the section "Dictionary of Statice File System Operations Commands", page 157.

Add ASYNCH DBFS PAGE Service Command

Add ASYNCH DBFS PAGE Service *host-name keywords*

Updates a host's namespace object to contain the ASYNCH-DBFS-PAGE service.

keywords :TCP Not Present

:TCP Not Present
{Yes No} If yes, no service entry is added for the TCP medium

Statice uses the ASYNCH-DBFS-PAGE service for communicating various signals and commands back to each of the client hosts, and hence should be present on all Statice clients. It need not be present on Statice servers however, unless they are clients to some other server.

This command adds the service-medium-protocol triplet for both the TCP and CHAOS mediums to the namespace object for the host. You should only need to perform this command once, when the file system is installed on a server. If the host does not support TCP, supply Yes to the :TCP Not Present keyword option.

Add DBFS PAGE Service Command

Add DBFS PAGE Service*host-name keywords*

Updates a host's namespace object to contain the DBFS-PAGE service.

keywords :TCP Not Present

:TCP Not Present
{Yes No} If yes, no service entry is added for the TCP medium.

Statice uses the DBFS-PAGE service uses for communicating database pages and requests over the network, and hence should be present on all Statice File System server hosts. It need not be present on client hosts, however.

This command adds the service-medium-protocol triplet for both the TCP and CHAOS mediums to the namespace object for the host. You should only need to perform this command once, when the file system is installed on a server. If the host does not support TCP, supply Yes to the :TCP Not Present keyword option.

Add Statice Partition Command

Add Statice Partition *file-system-name partition-pathname size*

Enables you to add partitions dynamically to a Statice file system (e.g. when it is running out of space).

file-system-name Name of a Statice file system that is stored on the local host.

partition-pathname Pathname of a partition; this pathname must name a FEPFS file on the local host (although the file need not exist).

size The size of the new partition, in blocks.

This command creates the new partition, allocates the space from the size given, and makes the new partition available to Statice for allocating.

This command may be given when a Statice server process receives a file system full error. For example, if a server process signals the following error, you can add a new partition and resume the operation:

```
Error: The File System "Squash" is full.

(FLAVOR:METHOD UFS::FIND-FREE-BLOCKS UFS:UFS-FILE-SYSTEM-MIXIN)
 proceed options...

 :Add Statice Partition (a file-system) SQUASH
 (the pathname of a file) FEP2:>Statice>SQUASH-part2.file.newest
 (Size in blocks [default 1000]) 1000

Updating file-system object SCRC|SQUASH in namespace... Done
```

After adding the partition, select the proceed option that resumes the operation.

Copy Statice Database Command

Copy Statice Database *from-database to-database keywords*

Copies all the pages of one database to the other (possibly new database) inside a transaction.

from-database Pathname of the database to copy.

to-database Pathname of the destination, where the database should be copied.

keywords :Copy Properties, :Create Directories, :Query

:Copy Properties {any combination of: Author, Comments, Creation Date} This indicates which properties should be copied to the new database(s). The default is Author and Creation Date.

:Create Directories
{Yes, Error, Query} Yes means that directories that do not exist should be created silently, Query will ask, and Error will cause an error if they do not exist.

:Query {Yes, No, Ask} Whether to ask before copying each file.

Create Statice File System Command

Create Statice File System *file-system-name keywords*

Creates a new Statice File System on the local host. You cannot use this command to create a file system on a remote host.

file-system-name A symbol naming the new file system.

keywords :Locally

:Locally {Yes, No} Whether to update only local namespace information (Yes), or to update the namespace database server as well (No). The default is No. See the section "The Locally Namespace Editor Command" in *Site Operations*.

The command displays an AVV menu in which you specify the names of various parameters. Above the menu, you will see a list of all the disk drives on the local host and the amount of free space available on each of them. The AVV menu asks for the following items:

Directory Partition: This entry specifies the FEPFS file in which the internal file system directory resides. Its size is determined by the number of directory entries which you specify in the Maximum Directory Entries field. The default file name for the directory partition is FEP*n*:>Statice>*fs-name*-part*m*.UFD. *n* is the highest mounted disk on the system, *fs-name* is the file-system name specified for the command, and *m* is the number of the partition.

Maximum Directory Entries:
This entry specifies the maximum number of databases which may reside in the file system at any time. Note that Statice always takes two of these entries for itself—one for the log file, and one for the Directory database. These entries are reusable, so if a database is deleted, using the Delete File command (in conjunction with a database pathname, not a FEPFS pathname), that entry in the file system directory is reusable for another database. On the Symbolics 36xx, there are 71 directory entries in each FEPFS block. The directory is organized as a hash file, so it's desirable to make the directory large enough that it's not densely filled.

Partition: These entries specify the partitions to be used for the file system. There may be as many partitions as you want, and they can live on any of the disks. In general, there should be as few partitions as possible in order to avoid disk fragmentation. The default pathname for a partition is FEP*m*:>Statice>*fs-name*-part*n*.file, where *m* is the highest mounted unit number, *fs-name* is the name of the file system, and *n* is the partition number in the ordering of all the partitions entered in the AVV menu.

Blocks: This entry specifies the number of blocks to allocate for the partition. When you enter a value for this field, the values in the available disk space headings will change accordingly to take into account how much of the free space you have allocated. You may click on None in this field to remove the partition the menu (and hence not include it as part of the file system when it is created).

When all the parameters have been entered, pressing `END` will cause the file system to be created. First, the file system object will be created in the namespace database (permanently, unless :Locally Yes was specified). Note that the messages printed by the command do not indicate whether the namespace was updated locally or globally. Second, all of the partitions are created in the FEPFS, and their :DONT-DELETE properties are set. You don't need to create any of the partitions yourself—this is done automatically for you, including the proper allocation of space. Third, the log file in the file system is initialized. Finally, the directory database is created.

Here's a sample run:

```
Command: Create Statice File System SQUASH
FEP0: 21464 Available (Originally: 21464 free, 88696/110160 used (81%))
FEP1:   137 Available (Originally: 137 free, 146743/146880 used (100%))
FEP2: 70727 Available (Originally: 71742 free, 38418/110160 used (35%))

Directory Partition: FEP2:>Statice>SQUASH.UFD
Maximum Directory Entries: 1000
Initial Log Size in Blocks: 500
Partition: FEP2:>Statice>part0.file.newest
  Blocks (None to remove): None  1000
Partition: FEP2:>Statice>part1.file.newest
  Blocks (None to remove): None  an integer

Creating file-system object SCRC|SQUASH in namespace... Done.
Initializing local UFS with associated directory structure... Done.
Creating local DBFS with associated directory structure... Done.
Initializing DBFS Directory database... Done.
```

Delete Statice File System Command

Delete Statice File System *file-system-name keyword*

Expunges an entire Statice file system, and removes all traces of it, including every database in it; this is a very dangerous command.

file-system-name A symbol naming a file system that is resident on the local host.

keywords :Locally

:Locally {Yes, No} Whether to update only local namespace information (Yes), or to update the namespace database server as well (No). The default is No. See the section "The Locally Namespace Editor Command" in *Site Operations*.

Because this command permanently removes the Statice File System, and all databases in it, it is a dangerous command and it asks for confirmation. If you answer Yes, the file system and all the databases in it are destroyed by removing the file system partitions from the FEP directory in which they were placed by the Create Statice File System command. The command destroys the file system partitions, even though they may have the :DONT-DELETE flag set for them in the FEPFS (the Create Statice File System command sets the :DONT-DELETE property for each of the partitions in a file system).

If you have done a complete backup dump, you can restore the contents of a deleted file system by using the Complete Restore command of the Statice File System Operations activity. If you have not done a complete backup, the data cannot be restored.

Dump Database Command

Dump Database *database-pathname destination-pathname*

Writes all the information in the database into a text file.

database-pathname A pathname indicating the location of a Statice database.

destination-pathname
A pathname of a file on any file system; this need not be stored on a Symbolics machine.

This command is useful for several purposes. It enables you to:

- Move a database from one place to another, over a channel that can handle only ordinary text.
- Store the contents of a database on some kind of storage medium (e.g. a particular tape format) that can handle only ordinary text.

- Edit the text file to reorganize the database.

We discuss the details of the Dump Database and Load Database commands elsewhere: See the section "High-level Dumper/Loader of Statice Databases", page 163.

Load Database Command

Load Database *database-pathname destination-pathname keywords*

Takes a text file produced by Dump Database, and makes a new database containing the same information.

database-pathname A pathname indicating the location of a Statice database.

destination-pathname
A pathname of a file on any file system; this need not be stored on a Symbolics machine.

keywords :If Exists

:If Exists {Error, Create} Specifies the action to be taken if the database specified by the *database-pathname* already exists. Error signals an error, and Create causes the old database to be erased and replaced by the database being loaded.

Unless you have edited the text file, Load Database makes an exact copy of the original database that was dumped, including keeping the unique ids the same.

We discuss the details of the Dump Database and Load Database commands elsewhere: See the section "High-level Dumper/Loader of Statice Databases", page 163.

Set Database Schema Name Command

Set Database Schema Name *pathname new-schema-name*

Informs the database that its schema name is now the given *new-schema-name*.

pathname A pathname indicating the location of a Statice database.

new-schema-name A symbol.

If you move a Statice program from one package to another, and the database already exists, it is necessary to use this command to update the database to inform it of the new schema name.

See the section "Warning About Changing the Package of a Statice Program", page 58.

Show All Statice File Systems Command

Show All Statice File Systems *namespace*

Lists all the file system objects in the namespace, and the host on which each one resides.

namespace A symbol that specifies a namespace in which to search. By default, all namespaces in the namespace search path are searched

Show Database Schema Command

Show Database Schema *pathname*

Prints the definition of the schema of the database specified by pathname. This command is useful if you see a database in a Statice file system and don't know what it is. It's also useful for seeing what indexes currently exist in a database.

Not all of the information from the template schema is stored in the database itself, so when Show Database Schema reconstructs the schema definition from the database, not all of the original information is recovered. Specifically:

The following attribute options are reconstructed: **:unique**, **:index**, **:index-average-size**, **:inverse-index**, **:inverse-index-average-size**, **:inverse-index-exact**, **:inverse-cached**, **:area**, **:attribute-set**, and **:no-nulls**.

The following attribute options are not reconstructed: **:cached**, **:initform**, **:inverse**, **:inverse-exact**, **:cluster**, **:accessor**, **:reader**, **:writer**, and **:read-only**.

The following entity-type options are reconstructed: **:area**, **:type-set**, **:multiple-index**, and **:multiple-index-exact**.

The following entity-type options are not reconstructed: **:conc-name**, **:constructor**, **:default-init-plist**, **:documentation**, **:init-keywords**, **:instance-variables**, and **:own-cluster**.

See the section "Examining the Schema of a Statice Database", page 110.

Show Statice Partitions Command

Show Statice Partitions *file-system-name*

Shows the amount of free space remaining in a file system's partition(s).

file-system-name Name of a Statice file system which resides on the local host.

This command may be done only for a file system stored on the local host.

For example:

```
Show Statice Partitions (a file-system) SQUASH

Partition                                   Free Space
FEP1:>squash>squash.file.newest          0/1056
FEP0:>squash>squash.file.newest          0/3000
FEP2:>Statice>SQUASH-part2.file.newest 54/1000
```

Update Database Schema Command

Update Database Schema *database-pathname*

Used when you have modified a schema; this command compares the template schema to the real schema in the database, and updates the real schema to match the template schema.

database-pathname A pathname indicating the location of a Statice database.

We discuss this subject in detail elsewhere: See the section "Modifying a Statice Schema", page 115.

7. Summary of Statice Operators

This section briefly describes the Statice operators. For complete documentation on each operator: See the section "Dictionary of Statice Operators", page 181.

Basic Use of Statice

statice:define-schema *schema-name entity-types*
Establishes a new schema and states which entity types comprise the schema.

statice:define-entity-type *type-name component-types attribute-clauses* &rest *options*
Defines a new type of Statice entity; also defines a constructor function that makes new entities and accessor functions that read and write information about an entity.

statice:make-database *pathname schema-name* &key *:databases (:if-exists* **:error***)*
Creates a new database and initializes it with a schema.

statice:with-database *(variable pathname)* &body *body*
Opens database indicated by *pathname*, binds the specified *variable* to an object representing the database, and executes the *body*.

statice:with-transaction *(*&key *(:automatic-retry* **'dbfs:restartable-transaction-abort***))* &body *body*
The dynamic extent of the **statice:with-transaction** form delimits a Statice *transaction*, which is a unit of work that is guaranteed to be *atomic*, *isolated*, and *persistent*.

statice:for-each *clauses* &body *body*
Selects a set of entities in a database based on criteria stated in its clauses.

statice:delete-entity *entity-handle* &optional *(database* **(statice:current-database)***)*
Removes the entity and all traces of it from the database.

Handling Set-valued Attributes

statice:add-to-set *set-valued-function-call value*
Adds the *value* to the set identified by *set-valued-function-call*.

statice:delete-from-set *set-valued-function-call value*
Removes *value* from the set identified by *set-valued-function-call*.

Dealing with Indexes

statice:make-index *function-name* &key *:index-average-size*
Creates an index for the Statice accessor function named by *function-name*.

statice:make-inverse-index *function-name* &key *:inverse-index-average-size :unique (:exact* **t***)*

Creates an inverse index for the accessor function whose name is *function-name*.

statice:make-multiple-index *list-of-function-names* &key *:unique (:exact* **t***)*
Creates a multiple index, for queries involving more than one function.

statice:delete-index *function-name*
Deletes the index for the Statice accessor function whose name is *function-name*.

statice:delete-inverse-index *function-name* &key *:exact*
Deletes the inverse index for the Statice accessor function whose name is *function-name*.

statice:delete-multiple-index *list-of-function-names* &key *(:exact* **t***)*
Deletes the multiple index for the Statice accessor functions that are listed in *list-of-function-names*.

statice:index-exists *function-name*
Returns **t** if an index exists on *function-name*, otherwise **nil**.

statice:inverse-index-exists *function-name* &key *(:exact* **t***)*
Returns **t** if an inverse index exists for the accessor *function-name*, otherwise **nil**.

statice:multiple-index-exists *list-of-function-names* &key *(:exact* **t***)*
Returns **t** if a multiple index exists for the functions in *list-of-function-name*s, otherwise **nil**.

Dealing with Strings or Arrays

statice:attribute-value-array-portion *entity-handle attribute from-start from-end into-array into-start*
Reads a portion of an array-valued attribute into a target array.

statice:attribute-value-length *entity-handle attribute*
Returns the length of the value of the *attribute* of the given *entity-handle*.

statice:set-attribute-value-array-portion *entity-handle attribute start end from from-start*
Writes from an array into a portion of an array-valued attribute.

statice:do-text-lines *(var string-valued-function-call* &key *(:delimiter* **'#\Return***) (:create-function* **'#'default-string-create-function***))* &body *body*
Allows a program to iterate over the actual lines of a string-valued attribute, thus eliminating the need to cons one big string and break it into lines.

Opening, Using, and Terminating Databases

statice:open-database *pathname* &optional *ok-if-not-found*
Opens the database indicated by *pathname*, if it's not already open.

statice:with-current-database *(database)* &body *body*
Makes the *database* object be the current database, for the (dynamic) extent of its *body*.

statice:current-database
Returns the current database.

statice:terminate-database *pathname*
Terminates the database stored in *pathname*, which is a database pathname.

Dynamic Statice Operations

statice:attribute-value *entity-handle attribute* &key *:into*
Reads the value of an attribute of an entity; this is the all-purpose accessor function that can choose the entity or attribute at run time.

statice:set-attribute-value-to-null *entity-handle attribute*
Sets the value of an attribute to null; this function can choose the entity or attribute at run time.

statice:inverse-attribute-value *entity-type attribute value* &key *(:exact* **t***)*
Returns the entity handle of the entity whose attribute is the given value; this is the all-purpose inverse reader function.

statice:attribute-value-null-p *entity-handle attribute*
Tests whether the value of the attribute of the given entity is the null value.

statice:make-entity *type-name* &rest *keywords-and-values*
Creates a new entity of the type *type-name* in the current database, initializes the entity according to the *keywords-and-values*, and returns the entity handle.

statice:for-each* *function entity-type* &key *:where :order-by :count (:database* **statice-model::*current-database***)*
This function is the dynamic version of the **statice:for-each** special form.

statice:add-to-set* *entity-handle attribute value*
This function is the dynamic version of the **statice:add-to-set** special form.

statice:delete-from-set* *entity-handle attribute value*
This function is the dynamic version of the **statice:delete-from-set** special form.

statice:count-entities* *entity-type* &key *:where (:database* **statice-model::*current-database***)*

Returns the number of entities of type *entity-type* in the specified database that match the **:where** specs.

statice:do-text-lines* *function entity-handle attribute* &key *(:delimiter* **'#\Return***) (:create-function* **#'default-string-create-function***)*
This is the dynamic version of **statice:do-text-lines**.

Miscellaneous

statice:with-cluster *(cluster-spec)* &body *body*
Defines the "current cluster" for the dynamic extent of the *body*: When you make a new entity within the dynamic extent of *body*, the entity is created inside the current cluster.

statice:*restart-testing*
This variable offers a way to help you test your code for robustness in the face of transaction aborting and restarting.

dbfs:set-buffer-replacement-parameters &key *(:page-pool-factor* **0.25***) (:page-pool-limit* **(* 1024 1024)***)*
Enables the user to limit the amount of virtual memory Statice will use as least-recently-used (LRU) buffer space.

Integrating Statice with a User Interface

statice:view-entity *stream pathname entity-handle* &optional *(setf-function* **#'statice::make-browser-attribute-value-setf***) values*
Displays an arbitrary entity in a window.

statice-utilities:entity-named-by-string-attribute (() &key *pathname type attribute restrictions*)
A presentation type for Statice entities that have simple string names, where the name is the value of a single-valued, string-typed attribute of the entity.

Defining New Statice Types

To define a new Statice type, you use **statice-type:define-value-type** (and, for some advanced applications, **statice-type:define-handler-flavor**), and define methods for some generic functions.

statice-type:define-value-type *type-name* &body *clauses*
Defines a new Statice value type.

statice-type:define-handler-flavor *handler-name* &body *clauses*
Used only in conjunction with **statice-type:define-value-type** and **:handler-finder**; this special form defines a storage handler flavor representing a new type.

statice-type:encode-value *handler value*
Methods for logical types should return the Lisp object that represents the *value* argument in the terms used by the underlying type indicated by *handler*.

statice-type:decode-value *handler value*

Methods for logical types should return the Lisp object that represents the *value* argument in terms of the type indicated by *handler*.

statice-type:read-value *handler addressor word-offset n-words-or-bit-offset*

Methods for physical types should make and return the Lisp representation of the value indicated by the arguments.

statice-type:value-equal *handler value addressor word-offset n-words-or-bit-offset*

Methods for physical types should return true if the value in the record is considered equal to the *value* argument. The null value must be considered equal only to the null value, and no other value.

statice-type:size-of-value *handler value*

Methods for physical types should return the number of words of a record that would be used to represent this value. Statice uses this method to determine how much space must be allocated to store a value into a record.

statice-type:write-value *handler value addressor word-offset n-words-or-bit-offset*

Methods for physical types should write the *value* into the portion of the record, or write an indication that the value is null.

statice-type:record-equal *handler addressor-1 word-offset-1 n-words-or-bit-offset-1 addressor-2 word-offset-2 n-words-or-bit-offset-2*

Methods are given two records, and must determine whether they are equal. Methods should return a true value if and only if the values stored in both records are not the null value and are equal to each other.

statice-type:record-compare *handler addressor-1 word-offset-1 n-words-or-bit-offset-1 addressor-2 word-offset-2 n-words-or-bit-offset-2*

Methods for physical types that are comparable receive two records, each holding a value; they must return one of the symbols **:lessp**, **:greaterp**, or **:equal**, based on the comparison of the records.

statice-type:value-compare *handler value addressor word-offset n-words-or-bit-offset*

Methods for physical types that are comparable receive a Lisp representation of a value and a a record holding a value. They return **:lessp**, **:greaterp**, **:equal**, or **statice-type:*null-value***, depending on how record compares to the Lisp value.

Examining Schemas

Statice offers a set of functions that enable you to examine any schema. They are not intended for use in ordinary Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. We summarize these functions below.

Getting a Schema Instance

The first step in examining a schema is getting a schema instance, which is an instance of either a real schema or a template schema.

statice:get-real-schema *pathname*
Returns an instance representing the real schema in the Statice database stored in the file indicated by *pathname*, which is a database pathname.

statice:get-template-schema *schema-name*
Returns an instance representing the template schema named *schema-name*.

If you need only the name of the schema for a particular database, it's not necessary to get a schema instance; instead, use the following function:

statice:get-real-schema-name *pathname*
Returns the name of the real schema in the Statice database stored in the file indicated by *pathname*, which is a database pathname. The result is a symbol, not a schema instance.

You can get a template entity type instance by using the following function:

statice:get-template-entity-type *entity-type-name*
Returns the entity type instance corresponding to *entity-type-name*, a symbol. (There is a separate function for getting a template entity type because template entity types can exist independent of any schema. As long as there is a **statice:define-entity-type** form, the entity type is defined, even if it's not a member of any schema.)

Operations on Schema Instances

statice:schema-name *schema*
Returns the name of the given *schema*.

statice:schema-types *schema*
Returns a list of entity types of the given *schema*.

Operations on Entity Type Instances

You get entity type instances by using **statice:schema-types**, or **statice:get-template-entity-type**. The following operations can be used on entity type instances:

statice:type-name *entity-type*
Returns the symbol that is the name of the given *entity-type*.

statice:type-parent-names *entity-type*
Returns the names of the parent types of the given *entity-type*.

statice:type-attributes *entity-type*
Returns the names of the attributes of the given *entity-type*.

statice:type-area-name *entity-type*
Returns the name of the area in which entities of the given *entity-type* are stored.

statice:type-set-exists *entity-type*
Returns true if a set exists for the given *entity-type*; otherwise, returns **nil**.

statice:type-multiple-indexes *entity-type*
Returns a list of multiple index instances of the given *entity-type*.

Operations on Attribute Instances

You get attribute instances by using **statice:type-attributes**. The following operations can be used on attribute instances:

statice:attribute-name *attribute*
Returns the name of the given *attribute*.

statice:attribute-function-name *attribute*
Returns the name of the reader function for the given *attribute*.

statice:attribute-type *attribute*
Returns the entity type of the given *attribute*.

statice:attribute-value-type *attribute*
Returns the value type of the given *attribute*.

statice:attribute-value-is-set *attribute*
Returns true if the attribute is set-valued; otherwise, returns **nil**.

statice:attribute-unique *attribute*
Returns true if the attribute's value is defined to be unique; otherwise, returns **nil**.

statice:attribute-read-only *attribute*
Returns true if the attribute is defined to be read-only; otherwise, returns **nil**.

statice:attribute-area-name *attribute*
Returns the name of the area (a symbol) in which values of the given *attribute* are stored.

statice:attribute-set-exists *attribute*
Returns true if a set exists for the given *attribute*; otherwise, returns **nil**.

statice:attribute-index-exists *attribute*
Returns true if an index exists for the given *attribute*; otherwise, returns **nil**.

statice:attribute-index-average-size *attribute*
Returns the average size defined for the attribute's index, or **nil** if no average size was specifed for the index.

statice:attribute-inverse-index-exists *attribute*
Returns true if an inverse index exists for the given *attribute*; otherwise, returns **nil**.

statice:attribute-inverse-index-exact-exists *attribute*
Returns true if an inverse exact index exists for the given *attribute*; otherwise, returns **nil**.

statice:attribute-inverse-index-average-size *attribute*
Returns the average size defined for the attribute's index, or **nil** if no average size was specifed for the index.

statice:attribute-no-nulls *attribute*
Returns true if **:no-nulls t** was specified for the given *attribute*; otherwise, returns **nil**.

statice:attribute-inverse-function-name *attribute*
Returns the name (a symbol) of the inverse function for the given *attribute*, or **nil** if the *attribute* has no inverse function.

Operations on Multiple Indexes

You get multiple index instances by using **statice:type-multiple-indexes**. The following operations can be used on multiple index instances:

statice:multiple-index-attribute-names *multiple-index*
Returns a list of names (symbols) of the attributes indexed by this multiple index.

statice:multiple-index-unique *multiple-index*
Returns true if the *multiple-index* is defined to be unique; otherwise, returns **nil**.

statice:multiple-index-case-sensitive *multiple-index*
Returns true if the *multiple-index* is defined to be case sensitive; otherwise, returns **nil**.

8. Dictionary of Statice Operators

statice:*restart-testing* *Variable*

Offers a way to help you test your code for robustness in the face of transaction aborting and restarting. Setting this variable will cause a restartable transaction abort to be signalled on every call to a Statice operator within a transaction, as well as when a transaction is about to commit. This is useful for testing whether a transaction causes fatal side effects. Note that **statice:for-each** is handled specially, depending on the value of **statice:*restart-testing***:

nil	Turns off restart testing mode; this is the default.
:all	Triggers aborts on every model-level call on every iteration of a **statice:for-each**. (Model level is the internal layer of software implementing the documented Statice entrypoints.)
:some	Triggers aborts only on the model-level calls in the first iteration of a **statice:for-each**.
other true value	Triggers no aborts during a **statice:for-each**.

Note that using restart testing mode is not guaranteed to find all problems caused by side effects within the code.

Note that using this mode slows down the performance significantly. We recommend that you not **setq** the value globally, which would affect all users of Statice in all processes. Instead, you should dynamically bind it around the code to be tested.

statice:add-to-set *set-valued-function-call value* *Special Form*

Adds the result of evaluating *value* to the set identified by *set-valued-function-call*. If the set already includes *value*, this will put a duplicate *value* into the set.

The argument *set-valued-function-call* is not evaluated; it simply identifies the set. It consists of a form that, if it were evaluated, would return a set, such as:

(*set-valued-function entity-handle*)

For example, the following form adds a course to Joe's set of courses:

```
(add-to-set (student-courses joe-cool) english-101)
```

Here, the value of **joe-cool** is an entity handle of type **student**; **student-courses** is an accessor of a set-valued attribute; and the value of **english-101** is a course. Note that **(student-courses joe-cool)** is not evaluated.

See the section "Set-Valued Attributes", page 22.

statice:add-to-set* *entity-handle attribute value* *Function*

The dynamic version of **statice:add-to-set**. The semantics and arguments are analogous to those of **statice:add-to-set**, but **statice:add-to-set*** is a function, whereas **statice:add-to-set** is a special form.

See the section "Dynamic Set Manipulation", page 102.

statice:attribute-area-name *attribute* *Function*

Returns the name of the area (a symbol) in which values of the given *attribute* are stored.

The *attribute* argument is an attribute instance, such as one of the attributes in the list returned by **statice:type-attributes**.

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:attribute-function-name *attribute* *Function*

Returns a symbol that is the name of the reader function for the given *attribute*.

The *attribute* argument is an attribute instance, such as one of the attributes in the list returned by **statice:type-attributes**.

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:attribute-index-average-size *attribute* *Function*

Returns the average size defined for the attribute's index, or **nil** if no average size was specifed for the index.

The *attribute* argument is an attribute instance, such as one of the attributes in the list returned by **statice:type-attributes**.

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:attribute-index-exists *attribute* *Function*

Returns true if an index exists for the given *attribute*; otherwise, returns **nil**.

The *attribute* argument is an attribute instance, such as one of the attributes in the list returned by **statice:type-attributes**.

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:attribute-inverse-function-name *attribute* *Function*

Returns the name (a symbol) of the inverse function for the given *attribute*, or **nil** if the *attribute* has no inverse function.

The *attribute* argument is an attribute instance, such as one of the attributes in the list returned by **statice:type-attributes**.

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:attribute-inverse-index-average-size *attribute* *Function*

Returns the average size defined for the attribute's index, or **nil** if no average size was specifed for the index.

The *attribute* argument is an attribute instance, such as one of the attributes in the list returned by **statice:type-attributes**.

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:attribute-inverse-index-exact-exists *attribute* *Function*

Returns true if an inverse exact index exists for the given *attribute*; otherwise, returns **nil**.

The *attribute* argument is an attribute instance, such as one of the attributes in the list returned by **statice:type-attributes**.

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:attribute-inverse-index-exists *attribute* *Function*

Returns true if an inverse index exists for the given *attribute*; otherwise, returns **nil**.

The *attribute* argument is an attribute instance, such as one of the attributes in the list returned by **statice:type-attributes**.

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:attribute-name *attribute* *Function*

Returns a symbol that is the name of the given *attribute*.

The *attribute* argument is an attribute instance, such as one of the attributes in the list returned by **statice:type-attributes**.

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:attribute-no-nulls *attribute* *Function*

Returns true if **:no-nulls t** was specified for the given *attribute*; otherwise, returns **nil**.

The *attribute* argument is an attribute instance, such as one of the attributes in the list returned by **statice:type-attributes**.

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:attribute-read-only *attribute* *Function*

Returns true if the attribute is defined to be read-only; otherwise, returns **nil**.

The *attribute* argument is an attribute instance, such as one of the attributes in the list returned by **statice:type-attributes**.

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:attribute-set-exists *attribute* *Function*

Returns true if a set exists for the given *attribute*; otherwise, returns **nil**.

The *attribute* argument is an attribute instance, such as one of the attributes in the list returned by **statice:type-attributes**.

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:attribute-unique *attribute* *Function*

Returns true if the attribute's value is defined to be unique; otherwise, returns **nil**.

The *attribute* argument is an attribute instance, such as one of the attributes in the list returned by **statice:type-attributes**.

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:attribute-value *entity-handle attribute* &key *:into* *Function*

An all-purpose reader function, which can be used to read the value of any attribute of any entity. You can use **setf** with **statice:attribute-value** to set the value of an attribute.

statice:attribute-value returns the same values as other reader functions:

First value	The value of the attribute, or **nil** if the attribute's value is the null value.
Second value	**t** if the attribute's value is not null and **nil** if the attribute's value is null.

entity-handle is an entity handle. *attribute* is a symbol, which can be either the name of the attribute, or the name of its reader function.

The keyword option is:

:into *string*	Enables you to read a string-valued attribute into the existing *string*. This means no new string is consed. The *string* should be long enough to hold the value. You can determine the length of the value with **statice:attribute-value-length**.

In the example below, the entity type is **person**; the attribute is **id-number**; and the value of the variable **george** is an entity handle:

```
;;; providing the name of the attribute
(attribute-value george 'id-number) => 123 and t

;;; providing the name of the reader function
(attribute-value george 'person-id-number) => 123 and t

;;; using the setf function
(setf (attribute-value george 'id-number) 72)
```

See the section "Dynamic Statice Accessor Functions", page 101.

statice:attribute-value-array-portion *entity-handle attribute from-start from-end into-array into-start* *Function*

Reads a portion of an array-valued attribute into a target array.

entity-handle	An entity handle.
attribute	A symbol, which can be either the name of the attribute, or the name of its reader function. The value of this attribute must be a vector or a string.
from-start	The starting position from which to read.
from-end	The ending position, where reading should stop.
into-array	The array in which to store the portion of the array-valued attribute.

into-start Indicates the position in the *into-array* where the writing should begin.

If the attribute is cached, the cache is updated. This function may be called outside a transaction if the attribute is cached.

statice:attribute-value-length *entity-handle attribute* *Function*

Returns the length of the value of the *attribute* of the given *entity-handle*. The type of the attribute must be string, vector, or symbol.

entity-handle is an entity handle. *attribute* is a symbol, which can be either the name of the attribute, or the name of its accessor function.

statice:attribute-value-is-set *attribute* *Function*

Returns true if the attribute is set-valued; otherwise, returns **nil**.

The *attribute* argument is an attribute instance, such as one of the attributes in the list returned by **statice:type-attributes**.

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:attribute-value-null-p *entity-handle attribute* *Function*

Tests whether the value of the attribute of the given entity is the null value. Returns **t** if it is the null value, otherwise **nil**.

entity-handle is an entity handle. *attribute* is a symbol, which can be either the name of the attribute, or the name of its reader function.

See the section "Dynamic Statice Accessor Functions", page 101.

statice:attribute-value-type *attribute* *Function*

Returns the value type of the given *attribute*.

The *attribute* argument is an attribute instance, such as one of the attributes in the list returned by **statice:type-attributes**.

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:count-entities* *entity-type* &key *:where (:database* **statice-model::*current-database***)* *Function*

Returns the number of entities of type *entity-type* in the specified database that match the **:where** specs.

entity-type is a symbol that names an entity type.

The keyword options are:

:where *where-specs* The syntax of *where-specs* is the same as in **statice:for-each***.

:database *database* Specifies the database in which to perform the count.

statice:current-database *Function*

Returns the current database.

statice-type:decode-value *handler value* *Generic Function*

Methods for logical types should return the Lisp object that represents the *value* argument in terms of the type indicated by *handler*.

This generic function is not intended to be called by users, but rather to be specialized with methods, when defining new Statice types. In general, each logical type is required to provide a method for **statice-type:decode-value**. See the section "Defining Logical Types", page 85.

For information on the arguments: See the section "Arguments to Methods for Defining New Statice Types", page 99.

Methods for **statice-type:decode-value** need not be conerned with null values: if the underlying type is holding a null value, these methods are not called.

When two elements of a logical type are compared to each other, for sorting or for "range" queries (that is, **:where** criteria involving comparisons), the underlying values are compared. Thus, the methods of encoding and decoding determine the equality predicate for values, as well as ordering of the values for comparison purposes and sorting purposes.

statice:define-entity-type *type-name component-types attribute-clauses* &rest *options* *Special Form*

Defines a new type of Statice entity. We give two examples here, and then discuss each piece of the syntax.

```
(define-entity-type person ()
  ((name string :unique t :no-nulls t :cached t
         :inverse person-named :inverse-index t)
   (id-number integer :unique t :read-only t)))

(define-entity-type student (person)
  ((dept department :inverse students-in-dept)
   (courses (set-of course) :index t :inverse course-students)
   (shirts (set-of shirt) :unique t :inverse shirt-owner)))
```

Statice automatically defines an *entity constructor function* for creating new entities of this type: See the section "Making New Statice Entities", page 12.

Statice automatically defines *accessor functions* for every attribute of the entity type. A reader function takes an entity as its argument and returns the value of an attribute. Statice also defines writer functions corresponding to the readers, so you can use **setf** with a reader function to set the value of

an attribute. See the section "Accessing Information in a Statice Database", page 14.

type-name A symbol naming the entity type.

component-types A list of names (symbols) of other entity types from which this type should inherit. See the section "Inheritance From Entity Types", page 27.

You can also include in this list the names of flavors to be components of this entity type. See the section "Mixing Flavors Into a Statice Entity Definition", page 109.

attribute-clauses This is a list, each of whose elements is an *attribute-clause*. Each *attribute-clause* is a list of the form:

(*name type attribute-options* . . .)

name is a symbol naming the attribute. This name must not be the name of an attribute of a component entity type.

type is the type of value to be stored in the attribute. For more information on the type of attributes:

See the section "Entity-Typed Attributes", page 21.
See the section "Set-Valued Attributes", page 22.

The *attribute-options* are:

:accessor *symbol*
Specifies that the name of the reader function for this attribute is the given *symbol*, and that the writer is to be called with the **setf** syntax with the reader named *symbol*. Overrides the default name and the **:conc-name** option for this attribute only. If you use **:accessor**, you cannot use **:reader** or **:writer** for the attribute.

:area *symbol*
Specifies an area, distinct from the default area, designated by the name *symbol*. All values of this attribute will be stored in this area. The name space of areas is independent of the names of entity types and attributes, so you can have both a type named **student** and an area named **student**. The area option is not inherited. See the section "Statice Type Sets, Attribute Sets, and Areas", page 127.

See the section "How to Control Type Sets, Attribute Sets, and Areas", page 131.

:attribute-set *boolean*

Specifies whether an attribute set exists for this attribute. The default is **t**. See the section "Statice Type Sets, Attribute Sets, and Areas", page 127.

See the section "How to Control Type Sets, Attribute Sets, and Areas", page 131.

:cached *boolean*

This option enables you to use the technique known as "taking snapshots" of the database. **:cached t** allocates an instance variable inside the flavor of the entity type. This instance variable is called the *cache slot* for the attribute. When the function for a **:cached** attribute is used *within* a transaction, it performs its normal accessor function, *and* it puts the value of the attribute into the cache slot. When the accessor function for a cached attribute is used from *outside* a transaction, it returns the contents of the cache slot. For more information: See the section "Taking Snapshots with the **:cached** Attribute Option", page 38.

:cluster *boolean*

This option can be used for only one attribute of this entity type, and only for an attribute that is single-valued and entity-typed. If **:cluster t** is specified for an attribute, then the **:own-cluster** entity option cannot be used in the same **statice:define-entity-type** form. **:cluster t** identifies the entity type being defined as a child in the context of clustering, and identifies this attribute as the one that designates the immediate parent. The goal is to achieve data locality, such that all children (entities of the type being defined by this **statice:define-entity-type** form) are placed in the cluster of the parent (the entity type that is the value of this attribute). The default is **:cluster nil**. For more information: See the section "Clustering Technique for Statice Databases", page 133.

:index *boolean*

If **t**, creates an index for the accessor of this attribute. See the section "Using Indexes to Increase Database Performance", page 45.

:index-average-size *integer-form*

Indicates your estimate of the average size of a set that would use this index; that is, how many results the set is likely to have. Such an estimate can sometimes speed up the query. The *integer-form* should evaluate to an integer. See the section "Statice Indexes", page 120.

:initform *form*

Provides a form to be the default initial value for the attribute. This form is evaluated inside the lexical environment of the containing **statice:define-entity-type** form. For information on exactly when the initform is used: See the section "The **:initform** Attribute Option", page 32.

:inverse *symbol*

Creates an inverse reader function named *symbol*. The argument of an inverse reader is the value of an attribute; the result is the entity handle for the entity that has the given value for this attribute. See the section "Accessing Information in a Statice Database", page 14.

When the attribute is entity-valued, an inverse writer is also defined: See the section "Inverse Writer Functions for Entity-typed Attributes", page 26.

:inverse-cached *boolean*

Has the same effect as the **:cached** option, but works for the inverse function. This option can be used only if you have specified **:inverse**, and for attributes that are entity-valued. There is no caching on a function whose argument is not an entity.

:inverse-exact *symbol*

Creates an exact inverse accessor function named *symbol*. The argument of an exact inverse accessor is the value of an attribute; the result is the entity handle for the entity that has the given value for this attribute. Whereas an inverse accessor uses regular comparison, an exact inverse accessor uses exact comparison. See the section "Regular Comparison Versus Exact Comparison", page 69.

See the section "Exact Inverse Accessor Functions", page 70.

:inverse-index *boolean*
If **t**, creates an index for the inverse accessor of this attribute. See the section "Using Indexes to Increase Database Performance", page 45.

:inverse-index-average-size *integer-form*
Indicates your estimate of the average size of a set that would use this inverse index; that is, how many results the set is likely to have. Such an estimate can sometimes speed up the query. The *integer-form* should evaluate to an integer. See the section "Statice Indexes", page 120.

:inverse-index-exact *boolean*
If **t**, creates an index for the exact inverse accessor of this attribute. See the section "Exact Indexes", page 70.

:no-nulls *boolean*
If **t**, means that values of the attribute are not allowed to be the null value; in other words, this is a "required attribute". For further information: See the section "The **:no-nulls** Attribute Option", page 31.

:read-only *boolean*
If **t**, means that no writer function is defined for this attribute. Once an entity has been created, the value of this attribute person never changes. For set-valued attributes, **:read-only t** also means that the special forms **statice:add-to-set** and **statice:delete-from-set** cannot be used. For an example: See the section "The **:read-only** Attribute Option", page 33.

:reader *symbol*
Specifies the name of the reader function for this attribute. Overrides the default name and the **:conc-name** option for this attribute only.

:writer *symbol*
Specifies the name of the writer function for this attribute. Overrides the default name and the **:conc-name** option for this attribute only. You call the writer function by using the **setf** syntax with the *symbol*.

:unique *boolean*
If **t**, then only one entity of this type can have a particular value for this attribute. However, for purposes of **:unique** checking, one null value *does*

not equal another null value; this means that several entities can have the null value as the value of this attribute. For more information: See the section "One-to-One, Many-to-One, and Other Relationships", page 24.

options A list of options pertaining to the entity type as a whole. The options include:

(:area *symbol***)**
Specifies an area, separate from the default area, designated by the name *symbol*. All entities of this entity type will be placed in this area. The name space of areas is independent of the names of entity types and attributes, so you can have both a type named **student** and an area named **student**. The area option is not inherited. See the section "Statice Type Sets, Attribute Sets, and Areas", page 127.

See the section "How to Control Type Sets, Attribute Sets, and Areas", page 131.

(:conc-name *symbol***)**
The *symbol* is used as the prefix of the names of the reader functions, instead of the default prefix, which consists of the entity type and the hyphen. See the section "The **:conc-name** Entity Type Option", page 36.

(:constructor *symbol***)**
The *symbol* is used as the name of the entity constructor function, instead of the default constructor name, which consists of the prefix **make-** followed by the the entity type name. See the section "Making New Statice Entities", page 12.

(:default-init-plist *plist***)**
Specifies a default-init-plist for the flavor that represents this entity type. See the section "**:default-init-plist** Option for **defflavor**" in *Symbolics Common Lisp Programming Constructs*.

(:documentation *string***)**
Specifies documentation for the flavor that represents this entity type. See the section "**:documentation** Option for **defflavor**" in *Symbolics Common Lisp Programming Constructs*.

(:init-keywords *symbols...* **)**
Specifies init-keywords for the flavor that repre-

sents this entity type. See the section "**:init-keywords** Option for **defflavor**" in *Symbolics Common Lisp Programming Constructs*.

(:instance-variables *spec-1 spec-2 ...***)**
Specifies instance variables to be included in the flavor being defined to represent this entity type. You can initialize, read, and write these instance variables (if you use the appropriate options in the instance variable *spec*). However, the values of these instance variables are maintained only in virtual memory, and are not stored in the Statice database. This advanced option can be used for customized caching schemes or for other purposes.

Each *spec* is a list of the form:

(*name option-1 option-2 ...*)

name is a symbol naming the instance variable. The *options* include:

:accessor *symbol*
:initform *form*
:reader *symbol*
:writer *symbol*

These options have the same semantics for instance variables as they do for attributes, with one exception. There are no accessors defined by default for these instance variables. If you want to read and/or write the value, you must specify **:accessor** option, or the **:reader** and/or **:writer** options. If you use **:accessor**, you cannot use **:reader** or **:writer** for that instance variable.

(:multiple-index (*attr-1 attr-2 ...***)** *option***)**
Creates a multiple index on the attributes. This multiple index is a compact table (a B+ tree) that associates tuples of attribute values with pointers to entities. The index entries are sorted by the values of *attr-1*, and groups of entries that all have the same value of *attr-1* are sorted within the group by *attr-2*, and so on.

There are two restrictions on multiple indexes:

1. The attributes must all be single-valued, not set-valued.

2. The attributes must all be from the entity type itself, not inherited from component entity types.

For the *option*, you can provide **:unique t** to impose uniqueness constraints on entity types. This states that no two entities can have both the same value for *attr-1* and the same value for *attr-2*, and so on. See the section "Multiple Indexes", page 51.

(:multiple-index-exact (*attr-1 attr-2 ...***) *option*)**
Just like the **:multiple-index** option, but the multiple index uses exact string comparison. See the section "Dealing with Strings in Statice", page 69.

(:own-cluster *boolean***)**
If **t**, then this entity type is defined as a parent in the context of clustering. Every time an entity of this type is created, it is placed in a new cluster. This means that a new page is allocated, and this page is empty except for the entity being created. Also, this page is not a member of any other cluster. The default for this option is **nil**. For more information: See the section "Clustering Technique for Statice Databases", page 133.

(:type-set *boolean***)**
If **t**, then this area has a type set; if **nil**, it does not. The default is **t**. See the section "Statice Type Sets, Attribute Sets, and Areas", page 127.

See the section "How to Control Type Sets, Attribute Sets, and Areas", page 131.

For related information:

See the section "Basic Concepts of Statice", page 3.
See the section "Defining a Statice Schema", page 7.
See the section "A More Complicated Schema: the University Example", page 20.
See the section "Order of Defining Pieces of a Schema", page 36.

statice-type:define-handler-flavor *handler-name* &body *clauses* *Special Form*
A special form whose syntax is just like that of **statice-type:define-value-type**, except that it does not accept the **:handler-finder** clause, and the name is the name of the flavor itself rather than the name of a type. It accepts all the other kinds of clauses, such as **:built-on** and **:comparable-p**. It should be used only in conjunction with **statice-type:define-value-type** and **:handler-finder**.

See the section "Flavors Representing a Statice Type", page 97.

statice:define-schema *schema-name entity-types* *Special Form*

Establishes a new schema and states the entity types that will comprise the schema. A schema consists of a single **statice:define-schema** form and a set of **statice:define-entity-type** forms.

schema-name is a symbol naming the new schema. *entity-types* is a list of symbols, each one the name of an entity type.

For example:

```
(define-schema university (person student graduate-student
                           shirt course instructor department))
```

statice:define-schema specifies a set of entity types. The entity types in the schema are the ones specified by define-schema, plus all the other entity types "referred to" by the ones specified, and the ones "referred to" by those, and so on. There are two ways that entity type A might "refer to" entity type B: if B is a component type of A, or if B is the type of an attribute of A. Statice starts with the list of entity types in the define-schema, and follows all the "refers to" links transitively to get the complete set of entity types.

The **statice:define-schema** form should explicitly mention all entity types that you intend to make available to other programs. If those entity types "refer to" other entity types that you do not include in the **statice:define-schema** form, those entity types will be a part of the schema, but not an externally-advertised part.

For example, you might provide an entity type A that's built on B, C, and D, and then offer A as a useful entity type for other peoples' schemas. People who use A would just list A in their own **statice:define-schema** forms, and would not have to be aware of B, C, and D. This promotes modularity.

For more information:

See the section "Basic Concepts of Statice", page 3.
See the section "Defining a Statice Schema", page 7.
See the section "Order of Defining Pieces of a Schema", page 36.

statice-type:define-value-type *type-name* &body *clauses* *Special Form*

Defines a new Statice value type. The *type-name* is a symbol naming the new type. The *clauses* are as follows:

(**:format** *format*) *format* can be **:logical**, indicating that this is a logical type; it can be **:variable**, indicating that this is a variable-format physical type; or it can be **:fixed**, indicating that this is a fixed-format physical type. See the section "Physical and Logical Statice Types", page 84.

(**:based-on** *presentation-type*)
This clause is used for logical types, to specify the *presentation-type* upon which this new type is based. The *presentation-type* must be understood by Statice before the new type can be used in a database.

(**:based-on-function** *function*)

This clause can be used instead of the **:based-on** clause in the definition of a logical type. The clause names a function. When any presentation type is given to Statice that is handled by this logical type definition, such as **(statice-type::member a b c)**, the function is called with that presentation type as its argument. The function must return the presentation type of the underlying type, e.g. **(statice-type::integer 0 (3))**.

(**:fixed-space** *size alignment*)

This clause is used for fixed-format physical types, to specify the amount of fixed space needed to hold a value. *size* is the size of the fixed space, in bits. *alignment* describes the alignment of the fixed space. If *alignment* is zero, the fixed space must be aligned on a word boundary. Otherwise, *alignment* should be a positive integer less than or equal to 16; it indicates that the bit position of the first bit in the field must be an integer multiple of the specified alignment. See the section "Defining a Fixed-Format Physical Type", page 92.

(**:comparable-p** *boolean*)

This clause is used for physical types; it indicates whether the values can be compared. The default is **statice-type::nil**. If *boolean* is **statice-type::t**, the type must provide methods for **statice-type:value-compare** and **statice-type:record-compare**.

(**:handler-finder** *arglist body*)

Defines a function called the *storage handler finder function*. It is called when a new attribute is made. Its first argument is the list of data arguments of the presentation type of the attribute. Its second argument is the no-nulls parameter of the new attribute. It must return a symbol that is the name of the storage handler flavor. When you use the **:handler-finder** clause, it should be the only clause. See the section "Flavors Representing a Statice Type", page 97.

See the section "Defining New Statice Types", page 84.

statice:delete-entity *entity-handle* &optional *(database* **(statice:current-database)***)* *Function*

Removes the entity specified by the *entity-handle* from the database. **delete-entity** also removes all traces of the entity from the database, which means:

- All functions that were previously valid for the entity are made invalid for that entity. Any attempt to use the deleted entity signals an error.

- If any single-valued attributes of any other entities previously had the deleted entity as a value, they now have the Statice null value.

- If any set-valued attribute of any other entity previously included the deleted entity as a member of its set, the entity is removed from the set.

The *database* argument specifies from which database the entity should be deleted. By default it is the current database.

There is a potential hazard when using **statice:delete-entity** within the body of a **statice:for-each**; if you delete an entity that the **statice:for-each** normally would have reached, but had not reached yet, an error will be signalled. That is, you can delete the entity that the current iteration of **statice:for-each** is working on, but you should not delete an entity that **statice:for-each** has not yet reached.

statice:delete-from-set *set-valued-function-call value* *Special Form*

Removes *value* from the set identified by *set-valued-function-call*. If the set includes *value* more than once, only one of the values is removed. **statice:delete-from-set** signals an error if the set does not include *value*.

The argument *set-valued-function-call* is not evaluated; it simply identifies the set. It consists of a form that, if it were evaluated, would return a set, such as:

(*set-valued-function entity-handle*)

For example, the following form deletes a course from Joe's set of courses:

```
(delete-from-set (student-courses joe-cool) english-101))
```

Here, the value of **joe-cool** is an entity handle of type **student**; **student-courses** is an accessor of a set-valued attribute; and the value of **english-101** is a course. Note that **(student-courses joe-cool)** is not evaluated.

See the section "Set-Valued Attributes", page 22.

statice:delete-from-set* *entity-handle attribute value* *Function*

The dynamic version of **statice:delete-from-set**. The semantics and arguments are analogous to those of **statice:delete-from-set**, but **statice:delete-from-set*** is a function, whereas **statice:delete-from-set** is a special form.

See the section "Dynamic Set Manipulation", page 102.

statice:delete-index *function-name* *Function*

Deletes the index for the Statice accessor function named *function-name*. Note: *function-name* must name an accessor function, not an inverse function. **statice:delete-index** can be used at any point within a transaction.

For example, this form deletes the index on the **courses** attribute of **student**:

```
(delete-index 'student-courses)
```

See the section "Using Indexes to Increase Database Performance", page 45.

statice:delete-inverse-index *function-name* &key *:exact* *Function*

Deletes the inverse index for the Statice accessor function whose name is *function-name*. Note: *function-name* must name an accessor function, not an inverse function. **statice:delete-inverse-index** can be used at any point within a transaction.

For example, this form deletes the inverse index on the **name** attribute of **person**:

```
(delete-inverse-index 'person-name)
```

The keyword option is:

:exact *boolean* If **t**, a case-sensitive inverse index is deleted; if **nil**, a case-insensitive inverse index is deleted. This is important for string-valued attributes only. See the section "Regular Comparison Versus Exact Comparison", page 69.

See the section "Using Indexes to Increase Database Performance", page 45.

statice:delete-multiple-index *list-of-function-names* &key *(:exact* **t***)* *Function*

Deletes the multiple index for the Statice accessor functions that are listed in *list-of-function-names*. Note: each function name in *list-of-function-names* must name an accessor function, not an inverse function. **statice:delete-multiple-index** can be used at any point within a transaction.

For example, to delete the multiple index on **title** and **dept**:

```
(delete-multiple-index '(course-title course-dept))
```

The keyword option is:

:exact *boolean* If **t**, a case-sensitive inverse index is deleted; if **nil**, a case-insensitive inverse index is deleted. This is important for string-valued attributes only. See the section "Regular Comparison Versus Exact Comparison", page 69.

See the section "Multiple Indexes", page 51.

statice:do-text-lines *(var string-valued-function-call* &key *(:delim-iter* **'#\Return***) (:create-function* **'#'default-string-create-function***))* &body *body* *Macro*

If a string-valued attribute is used for storing very long text strings which consist of multiple lines, the programmer may not want to cons the whole string if it will later be broken down into many substrings. **statice:do-text-lines** allows a program to iterate over the actual lines (or other kinds of substrings as defined by some delimiter character).

This macro iterates over all the lines of the value specified by *string-valued-function-call*, binding *var* to each line of the value. The following example iterates over all the text-entity entities in a database and prints out

individual text lines.

```
(define-entity-type text-entity ()
  ((lines string)))

(with-database (db pathname)
  (with-transaction ()
    (for-each ((ent text-entity))
      (do-text-lines (tl (text-entity-lines ent))
      (princ tl)
      (terpri))))
```

The arguments and keywords are:

string-valued-function-call
: A function call of a reader whose value is a string.

var
: A symbol.

:delimiter
: Specifies what character should be used as the delimiter for the lines. The default is **#\return**.

:create-function
: Specifies the function to use for creating strings. The function should take two arguments: a length, and an argument specifying whether the string needs to be able to hold fat characters or not. We show the default function below.

The individual lines as passed to the **:create-function** program do not contain the delimiter character. The default function is defined as follows:

```
;;; Default string creator for do-text-lines and do-text-lines*
(defun default-string-create-function (length thin-p)
  (make-string length :element-type
                      (if thin-p 'string-char 'character)))
```

There is a block around the body of the **statice:for-each**, with the tag being **nil**. This means you can use **(return** *form***)** to exit from the **statice:do-text-lines**.

statice:do-text-lines* *function entity-handle attribute* &key *(:delimiter* **'#\Return***) (:create-function* **#'default-string-create-function***)* *Function*

The dynamic version of **statice:do-text-lines**. The syntax is somewhat different. **statice:do-text-lines*** takes the following arguments and keywords:

function
: A function that takes one argument, the text line, and processes it in whatever way is desired. This argument takes the place of the *body* of **do-text-lines**.

entity-handle
: The entity handle of an entity. The value of the *attribute* of this entity is the string which is given to the *function*.

attribute May be the symbol that names the attribute, or the function object obtained by the schema querying functions.

:delimiter Same as for **statice:do-text-lines**.

:create-function Same as for **statice:do-text-lines**.

statice-utilities:entity-named-by-string-attribute (() &key *pathname type attribute restrictions*) *Presentation Type*

A presentation type for Statice entities that have simple string names, where the value of a single-valued, string-typed attribute of the entity is considered the name of the entity. Most applications are expected to make a presentation type that is an abbreviation for this presentation type.

The first three data arguments are all mandatory; the last one is optional.

pathname is the pathname of the database.

type is the name of the entity type.

attribute is the name of the single-valued, string-typed attribute of the type that serves to name entities.

restrictions is a list of criteria, just like the **:where** argument to **statice:for-each***, and it means that only the subset of entities that pass all of these criteria are considered to be part of the set.

For example, here is an entity type that has a simple string name, namely the person entity type from the university example:

```
(define-entity-type person ()
  ((name string :unique t :no-nulls t :cached t
        :inverse person-named :inverse-index t)
   (id-number integer :unique t :read-only t)))
```

If we want a presentation type that will prompt the user for the name of a person in the database in ***university-pathname***:

```
`((statice-utilities:entity-named-by-string-attribute)
   :pathname ,*university-pathname* :type person :attribute name)
```

Since that's rather verbose, you might want to make an abbreviation:

```
(define-presentation-type name-in-the-university (() &key pathname)
   :abbreviation-for
     `((statice-utilities:entity-named-by-string-attribute)
        :pathname ,pathname :type person :attribute name))
```

Having defined this abbreviation, you can do things like the following:

```
(accept `((name-in-the-university) :pathname ,*university-pathname*))
```

The presentation type implements completion efficiently, by using Statice queries in the database, rather than reading all the names out of the database.

These examples are in the file **sys:statice;examples;university-example.lisp**.

statice:for-each *clauses* &body *body* *Special Form*

Selects a set of entities in a database. This is the first step in many Statice programs; once the entities have been located in the database and made available to the program, you can operate on them in any way you choose. Within the body of the **statice:for-each**, you can access each entity by means of a variable.

Note: **statice:for-each** is a macro. If you change the definitions of any of the types that affect a **statice:for-each**, you must recompile all functions that use those types.

In the simplest use of **statice:for-each** you iterate over all entities of a given type. See the section "Iterating Over an Entity Type", page 16. In the **bank-total** function, **statice:for-each** establishes a variable called **a**, and binds **a** successively to entity handles for each **account** entity in the database. It runs the body once for each entity, and the body accumulates the sum of the balances.

```
(defun bank-total ()
  (with-database (db *bank-pathname*)
    (with-transaction ()
      (let ((result 0))
        (for-each ((a account))
          (incf result (account-balance a)))
        result))))
```

You can use **statice:for-each** to specify filters to select entities that satisfy one or more criteria. A **:where** clause restricts the **statice:for-each**: instead of iterating over every member of the set, it only iterates over those for which the criterion of the **:where** clause is true. For example, the following **statice:for-each** clause selects a person whose id-number is 100:

```
(for-each ((p person) (:where (= (person-id-number p) 100)))
```

The **:where** clause of **statice:for-each** is the basic kind of associative database query provided by Statice. **statice:for-each** can iterate not only over the set of all entities of some type, but also over the value of a set-valued function. A **:where** clause can be used in the same way no matter what set is being iterated over.

For examples of using **statice:for-each**: See the section "Querying a Statice Database with **statice:for-each**", page 41.

Syntax of statice:for-each

The notation conventions used here are the same as the modified BNF described in *Common Lisp the Language*, page 8.

`(for-each ({`*variable-spec*`}+ {`*clause*`}*) . `*body*`)`

clause ::= *count-clause* | *where-clause* | *order-by-clause* | *database-clause*

variable-spec ::= (*variable set-expression*)

variable ::= *symbol*

set-expression ::= *type-name* | *set-function-form*

type-name ::= *symbol*

set-function-form ::= *(set-function-name any-lisp-form)*

count-clause ::= (:count *integer*)

where-clause ::= (:where *criterion*) | (:where (and {*criterion*}+))

criterion ::=
(*rev-op attribute-spec any-lisp-form*) |
(*rev-op any-lisp-form attribute-spec*) |
(*non-rev-op any-lisp-form attribute-spec*) |
(*equality-op variable attribute-spec*) |
(*equality-op attribute-spec variable*) |
(*rev-op attribute-spec attribute-spec*) |
(typep *attribute-spec any-lisp-form*) |
(null *attribute-spec*)

attribute-spec ::=
(*single-function-name variable*) |
(:any (*set-function-name variable*) |
variable

rev-op ::= *equality-op* | < | > | ≤ | ≥ | <= | >= |
string< | string> | string≤ | string≥ | string≤ | string<= |
string-lessp | string-greaterp | string-not-lessp | string-not-greaterp |
char< | char> | char≤ | char≥ | char≤ | char<= |
char-lessp | char-greaterp | char-not-lessp | char-not-greaterp

equality-op ::= eq | eql | equal | string-equal | = | string=

non-rev-op ::=
string-search | string-search-exact |
string-prefix | string-prefix-exact

order-by-clause ::=
(:order-by {(*single-function-name variable*) *direction*}+) |
(:order-by {*variable direction*}+)

direction ::= ascending | descending

database-clause ::= (:database *any-lisp-form*)

Additional rules:

A *clause* has a number of *variable-set-expression-clauses*; the cross-product of them indicates the set over which to iterate. A *clause* can have zero or one *count-clause*, zero or one *where-clause*, zero or one *order-by-clause*, and zero or one *database-clause*. The *variable-set-expression-clauses* must come before the other kinds of *clauses*.

set-function-name means the name of a set-valued accessor function. *single-function-name* means the name of a single-valued accessor function. *function-name* means the name of an accessor function that is either set-valued or single-valued.

When several criteria are combined with **and**, it is valid for more than one criterion to mention the same function. This is particularly useful with pairs of inequality operators, to examine a range.

The *variable* in a *criterion* or *order-by-clause* must be the same as the top-level variable of iteration. The *single-function-name* in a *criterion* or *order-by-clause* must be a single-valued function of the top-level *type-name*.

Details of the Execution:

First, all of the Lisp forms that preceed the body are evaluated. The order in which those evaluations takes place is not defined. These evaluations are all outside the lexical scope of the variables of iteration.

The particular database to which the functions and types refer is the value of the *database-clause*, which must be a database object (or else an error is signalled). If there is no *database-clause*, the current database is used.

If there is only one *variable-spec*, the body is executed repeatedly, with the value of *variable* taking on the value of each element of *set-expression*, restricted to the subset defined by the *where-clause* if any, in the order specified by the *order-by-clause* if any, stopping after the number of times specified in the *count-clause* if any. If *set-expression* is a *type-name*, then the values are all entity handles. If *set-expression* is a *set-function-form*, then the values could be entity handles, or data values (numbers, strings, etc).

If there are two or more *variable-spec*s, the body is executed once for every possible combination of values of each set expression, i.e. for the Cartesian product the values, restricted to the subset defined by the *where-clause* if any, in the order specified by the *order-by-clause* if any, limited by the *count-clause* if any.

A *count-clause* limits the number of entities for which the body is called. If the integer after **:count** is one, the body is called for only one entity. If only one entity is desired, the *count-clause* should reduce consing and improve query performance when you want to just find the first one of something.

There is a block around the body of the **statice:for-each**, with the tag being **nil**. This means you can use **(return** *form***)** to exit from the **statice:for-each**.

There is an ambiguity in the syntax of a *criterion* whose operator is a *rev-op*, because sometimes either subform could be interpreted as an *attribute-spec*. In such a case, the first subform is taken to be the *attribute-spec*, and the second subform is taken to be the *any-lisp-form*.

Performance Implications of statice:for-each

An individual program can specify filtering restrictions by using **when** forms in the body of a **statice:for-each** special form instead of using the **:where** clause. However, in most cases Statice can implement the restrictions much more efficiently, by using indexes, if you use the **:where** clause. This is all done automatically when you use **statice:for-each**.

When a set-valued function is called as a Lisp function in the ordinary way, it creates and returns a list of all the values of the set (in this case, a list of entity handles, each of which represents a course). However, when a set-valued function is used inside a **statice:for-each**, no actual list is created. Since this avoids "consing", it may result in better performance, particularly if the set is large.

:where Criteria

A **:where** clause can have one criterion, or many criteria grouped by **and**. If there are many criteria, all must be true for the body to be called.

The following sequence of examples of **:where** criteria illustrate all of the acceptable forms of criteria.

- *(rev-op attribute-spec any-lisp-form)*

```
(for-each ((i instructor)
           (:where (> (instructor-salary i) this-much)))
  (push (person-name i) instructors))
```

  rev-op is **>**, *attribute-spec* is **(instructor-salary i)**, and *any-lisp-form* is **this-much**. For a particular value of **i** under consideration, the criterion is true if the value of the **salary** attribute of **i** is greater than the value of the form **this-much**.

- *(rev-op any-lisp-form attribute-spec)*

```
(for-each ((i instructor)
           (:where (< this-much (instructor-salary i))))
  (push (person-name i) instructors))
```

  Now *rev-op* is **<**, and the arguments have been reversed. This is equivalent to the previous example.

- *(non-rev-op any-lisp-form attribute-spec)*

```
(for-each ((f faculty)
           (:where (string-search "a" (employee-office f))))
  (push (person-name f) result))
```

string-search is a *non-rev-op*, which means that the *any-lisp-form* must come before the *attribute-spec*.

- (*equality-op variable attribute-spec*)

```
(for-each ((s student) (d department)
           (:where (eq d (student-dept s))))
  (push (list (person-name s) (department-name d)) results))
```

There are two *variable-spec*s in this example. The *equality-op* is **eq**. The *variable* is **d**, which is one of the variables of iteration. The *attribute-spec* is **(student-dept s)**, which refers to the other variable of iteration.

- (*equality-op attribute-spec variable*)

```
(for-each ((s student) (d department)
           (:where (eq (student-dept s) d)))
  (push (list (person-name s) (department-name d)) results))
```

Now the arguments have been reversed. This is equivalent to the previous example.

(*rev-op attribute-spec attribute-spec*)

```
(for-each ((s student) (d department)
           (:where (eq (student-advisor s) (department-head d))))
  (push (list (person-name s) (department-name d)) results))
```

The *rev-op* is **eq**, and there are two *attribute-spec*s: **(student-advisor s)** and **(department-head d)**. This query finds all pairs of a student and a department such that the student's advisor is the head of the department.

- (typep *attribute-spec any-lisp-form*)

```
(for-each ((g graduate)
           (:where (typep (student-advisor g) 'faculty)))
  (push (person-name g) result))
```

When the operator is **typep**, the *attribute-spec* is first and the *any-lisp-form* is second. This query collects the names of all graduates whose advisor is a faculty member.

- `(null` *attribute-spec*`)`

```
(for-each ((p person)
           (:where (null (person-id-number p))))
  (push p no-ids))
```

When the operator is **null**, there's only one argument. This query collects a list of people who have no ID number.

- *(rev-op attribute-spec any-lisp-form)*

```
(define-entity-type host ()
  ((names (set-of string))
  ...))

(for-each ((n (host-names host-1))
           (:where (string-greaterp n "M")))
  (push n x))
```

This is an example of a familiar criterion syntax, where the *attribute-spec* is a *variable* instead of a list. An *attribute-spec* can only be a *variable* when the variable is iterating over an attribute value whose type is a set of data values (not a set of entities). In this example, *rev-op* is **string-greaterp**, *attribute-spec* is **n**, and *any-lisp-form* is **"M"**. The query collects all the names of **host-1** that are alphabetically greater than "M".

statice:for-each* *function entity-type* &key *:where :order-by :count (:database* **statice-model::*current-database***)* *Function*

The dynamic version of **statice:for-each**. **statice:for-each*** is a function, whereas **statice:for-each** is a special form. **statice:for-each*** lets you specify at run time which entity type to iterate over, what criteria to use, and so on.

function — This function is called once for every entity that **statice:for-each*** finds. The function is called on one argument, the entity handle, and its returned value is ignored. It's analogous to the body of the **statice:for-each** special form.

entity-type — The name of an entity type.

keywords — If no keywords are supplied, the function is called once for each entity of the entity type. The keywords have the same semantics as their counterparts in **statice:for-each**.

For example, the following form builds a list, in the variable **result**, of all people whose name follows **"r"**, sorted by the person's **ssn**:

```
(for-each* #'(lambda (person)
                   (push (person-name person) result))
               'person
               :where '((person-name string-greaterp "r"))
               :order-by '(person-ssn descending))
```

In Statice 2.0, **statice:for-each*** does not support the full set of functionality as does **statice:for-each**. See the section "Limitations on **statice:for-each*** in Statice 2.1".

See the section "Dynamic Statice Queries", page 103.

statice:get-real-schema *pathname* *Function*

Returns an instance representing the real schema in the Statice database stored in the file indicated by *pathname*, which is a database pathname.

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:get-real-schema-name *pathname* *Function*

Returns the name of the real schema in the Statice database stored in the file indicated by *pathname*, which is a database pathname. The result is a symbol, not a schema instance. Thus, given an existing database, you can use **statice:get-real-schema-name** to find out what symbol was given to **statice:make-database** when the database was created, without going through the full expense of **statice:get-real-schema**.

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:get-template-entity-type *entity-type-name* *Function*

Returns the entity type instance corresponding to *entity-type-name*, a symbol.

There is a separate function for getting a template entity type because template entity types can exist independent of any schema. As long as there is a **statice:define-entity-type** form, the entity type is defined, even if it's not a member of any schema. Real entity types, on the other hand, always reside within databases, and every database has a real schema. Thus, there is no need for a function called "**statice:get-real-entity-type**"; you can get real entity types from the real schema instance.

statice:get-template-schema *schema-name* *Function*

Returns an instance representing the template schema named *schema-name*, a symbol.

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a

browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:index-exists *function-name* *Function*

Returns **t** if an index exists on *function-name*, otherwise **nil**. Note: *function-name* must name an accessor function, not an inverse function.

For example, the following form asks whether there is an index on the **courses** attribute of **student**:

```
(index-exists 'student-courses)
```

See the section "Using Indexes to Increase Database Performance", page 45.

statice:inverse-attribute-value *entity-type attribute value* &key *(:exact* **t***)* *Function*

Returns the entity handle of the entity whose attribute is the given value; this is the all-purpose inverse reader function.

entity-type is the name of the entity type of the attribute. *value* is the value whose inverse is desired. *attribute* is a symbol, which can be either the name of the attribute, or the name of its reader function.

For example:

```
;;; providing the name of the attribute
(inverse-attribute-value 'person 'name "george")

;;; providing the name of the reader function
(inverse-attribute-value 'person 'person-name "george")
```

The keyword option is:

:exact *boolean* Controls whether the comparison of the given *value* to the values of attributes of entities is case-sensitive or not. A value of **t** does a case-sensitive comparison, while **nil** does a case-insensitive comparison. This is important only for string-valued inverse functions. See the section "Regular Comparison Versus Exact Comparison", page 69.

See the section "Dynamic Statice Accessor Functions", page 101.

statice:inverse-index-exists *function-name* &key *(:exact* **t***)* *Function*

Returns **t** if an inverse index exists for the accessor *function-name*, otherwise **nil**. Note: *function-name* must name an accessor function, not an inverse function.

The keyword option is:

:exact *boolean* If **t**, a case-sensitive inverse index is checked for existence; if **nil**, a case-insensitive inverse index is checked for existence. This is important for string-valued at-

tributes only. See the section "Regular Comparison Versus Exact Comparison", page 69.

See the section "Using Indexes to Increase Database Performance", page 45.

statice:make-database *pathname schema-name* &key *:databases (:if-exists* **:error***)* *Function*

Creates and returns a new Statice database in the file designated by the *pathname* argument. The database is initialized with a new schema, indicated by *schema-name*.

schema-name is a symbol naming a schema definition, which was initially created by **statice:define-schema**. Any types defined by **statice:define-entity-type** that correspond to this schema are also part of the schema definition. After **statice:make-database** is run, the new database contains this schema, and all the types are initially empty. That is, there are no entities of these types.

The keyword options are:

:databases (Ignored)

:if-exists Controls what happens if a database already exists with that pathname; the possible values are **:error** and **:create**. When **:if-exists** is **:error** (the default) an error is signaled; this is the **dbfs:file-already-exists** error. When **:if-exists** is **:create**, a new database is created, and all data in the existing database is erased. The data cannot be recovered except by a system backup.

For example:

```
;;; Save a pathname that refers to database
(defvar *bank-pathname* #p"beet:>finance>bank")

(defun make-bank-database ()
  (make-database *bank-pathname* 'bank))
```

See the section "Making a Statice Database", page 8.

statice:make-entity *type-name* &rest *keywords-and-values* *Function*

Creates a new entity of the type *type-name* in the current database, initializes the entity according to the *keywords-and-values*, and returns the entity handle. **statice:make-entity** is the all-purpose entity constructor function. The *keywords-and-values* are the same as the arguments to the entity constructor function of that entity type; that is, there is a keyword for every attribute of the entity type. The names of the keywords are the same as the names of the attributes, but in the keyword package. The value following the keyword is used to initialize the attribute.

For example, if the entity type named **person** has attribute named **name** and **id-number**, the following form creates a new person and initializes the

person's name and id-number:

```
(make-entity 'person :name "Beth" :id-number 23)
```

For more information:

See the section "Dynamic Entity Creation", page 102.
See the section "Making New Statice Entities", page 12.
See the section "The **:initform** Attribute Option", page 32.

statice:make-index *function-name* &key *:index-average-size* *Function*

Creates an index for the Statice accessor function named *function-name*. Note: *function-name* must name an accessor function, not an inverse function. **statice:make-index** can be used at any point within a transaction.

For example, the following form makes an index on the **shirts** attribute of **student**:

```
(make-index 'student-shirts)
```

The keyword option is:

:index-average-size *integer*
: Gives Statice an estimated size of the indexed set. The integer should be an estimate of how many results the query is likely to have.

In general, it is meaningful to make indexes for accessors that return the value of set-valued attributes. For accessors of single-valued attributes, there is no need for an index, since Statice can obtain the value directly without searching. (There is one case in which it does make sense to create an index for an accessor of a single-valued attribute, involving areas. See the subheading Areas and Indexes in the section "Statice Type Sets, Attribute Sets, and Areas".)

If a large number of entities of the type already exist, it might take some time for Statice to make the index.

See the section "Using Indexes to Increase Database Performance", page 45.

statice:make-inverse-index *function-name* &key *:inverse-index-average-size :unique (:exact* **t***)* *Function*

Creates an inverse index for the Statice accessor function named *function-name*. Note: *function-name* must name an accessor function, not an inverse accessor function. You can make an inverse index on a function even if there is no inverse accessor function specified in the **statice:define-entity-type** form. **statice:make-inverse-index** can be used at any point within a transaction.

For example, the following form makes an inverse index on the **size** attribute of **shirt**:

```
(make-inverse-index 'shirt-size)
```

The keyword options are:

:inverse-index-average-size *integer*
Gives Statice an estimated size of the indexed set. The integer should be an estimate of how many results the query is likely to have.

:unique *boolean*

:exact *boolean* Controls whether the index is case-sensitive or not. A value of **t** makes a case-sensitive index, while **nil** makes a case-insensitive index. This is important only for string-valued inverse functions. See the section "Regular Comparison Versus Exact Comparison", page 69.

If a large number of entities of the type already exist, it might take some time for Statice to make the index.

See the section "Using Indexes to Increase Database Performance", page 45.

statice:make-multiple-index *list-of-function-names* &key *:unique* *(:exact* **t***)* *Function*

Creates a multiple index, for queries involving more than one function. Each element of *list-of-function-names* is the name of a Statice accessor function.

You can create a multiple index when all of the following requirements are met:

- All functions listed are single-valued functions.
- All functions listed are in the same area as the type.
- None of the functions is an inverse function.
- At least two functions are listed.
- The attributes must all be from the entity type itself, not inherited from component entity types.

For example, the following form makes a multiple index on the **size** and **color** attributes of the **shirt** entity type:

```
(make-multiple-index '(shirt-size shirt-color))
```

The keyword options are:

:unique *boolean* If **t**, this imposes uniqueness constraints on entity types: no two entities can have both the same value for all of the attributes indicated by the function names. That is, if this is a multiple index on three function names, then the uniqueness contraint ensures that no two entities have all the same values for the three attributes. If **nil**, there is no uniqueness constraint. (Note that for purposes of **:unique** checking, one null value *does not equal* another null value; this means that several entities can have the null value as the value of this attribute.)

:exact *boolean* Controls whether the index is case-sensitive or not. A value of **t** makes a case-sensitive index, while **nil** makes a case-insensitive index. This is important only for string-valued inverse functions. See the section "Regular Comparison Versus Exact Comparison", page 69.

See the section "Multiple Indexes", page 51.

statice:multiple-index-attribute-names *multiple-index* *Function*

Returns a list of names (symbols) of the attributes indexed by this multiple index.

The *multiple-index* argument is a multiple index instance, such as one of the multiple indexes in the list returned by **statice:type-multiple-indexes**.

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:multiple-index-case-sensitive *multiple-index* *Function*

Returns true if the *multiple-index* is defined to be case sensitive; otherwise, returns **nil**.

The *multiple-index* argument is a multiple index instance, such as one of the multiple indexes in the list returned by **statice:type-multiple-indexes**.

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:multiple-index-exists *list-of-function-names* &key *(:exact* **t***)* *Function*

Returns **t** if a multiple index exists for the functions in *list-of-function-name*s, otherwise **nil**. Each function name in the list should be a symbol that names a Statice accessor function (not an inverse function).

For example, the following form asks whether there is a multiple index on the **title** and **dept** attributes of **course**:

```
(multiple-index-exists '(course-title course-dept))
```

The keyword option is:

:exact *boolean* If **t**, a case-sensitive multiple index is checked for existence; if **nil**, a case-insensitive multiple index is checked for existence. This is important for string-valued attributes only. See the section "Regular Comparison Versus Exact Comparison", page 69.

See the section "Multiple Indexes", page 51.

statice:multiple-index-unique *multiple-index* *Function*

Returns true if the *multiple-index* is defined to be unique; otherwise, returns **nil**.

The *multiple-index* argument is a multiple index instance, such as one of the multiple indexes in the list returned by **statice:type-multiple-indexes**.

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:open-database *pathname* &optional *ok-if-not-found* *Function*

Opens the database indicated by *pathname*, if it's not already open. **statice:open-database** returns the database object.

Along with **statice:with-current-database**, **statice:open-database** can be considered a primitive underlying **statice:with-database**. In some cases, users can use these primitives to achieve greater speed, or to deal with more than one database (such as when copying data from one to another). For details: See the section "Opening and Terminating Databases", page 72.

statice-storage:read-multiple-record-word *record-addressor start-index end-index* &key *:into :into-start* *Generic Function*

Reads a contiguous subsequence of words from the record into an array. The array is made and returned on the data stack, so the caller is required to provide a **sys:with-data-stack** special form around the dynamic extent of the use of the array.

This generic function is used only in the methods for defining physical types: See the section "Defining Physical Types", page 88.

statice-storage:read-record-word *record-addressor index* &key *:buffer-p* *Generic Function*

Reads the word specified by *index* from the record specified by *record-addressor*, and returns it. Always returns a signed 32-bit integer.

You can use **statice-type::setf** on this function, which has the same effect as **statice-storage:write-record-word**.

This generic function is used only in the methods for defining physical types: See the section "Defining Physical Types", page 88.

statice-type:read-value *handler addressor word-offset n-words-or-bit-offset* *Generic Function*

Methods for physical types should make and return the Lisp representation of the value indicated by the arguments.

This generic function is not intended to be called by users, but rather to be specialized with methods, when defining new Statice types. Each physical type must provide a method for this generic function. See the section "Defining a Variable-Format Physical Type", page 89.

See the section "Defining a Fixed-Format Physical Type", page 92.

For information on the arguments: See the section "Arguments to Methods for Defining New Statice Types", page 99.

The *addressor* is read-only (or, in any event, not guaranteed to be writable), and methods must not write into the record.

statice-type:record-compare *handler addressor-1 word-offset-1 n-words-or-bit-offset-1 addressor-2 word-offset-2 n-words-or-bit-offset-2* *Generic Function*

Methods for physical types that are comparable receive two records, each holding a value; they must return one of the symbols **:lessp**, **:greaterp**, or **:equal**, based on the comparison of the records. Null values are considered to be equal to each other, and greater than all other values. **statice-type:record-compare** avoids allocating Lisp storage for rational numbers.

This generic function is not intended to be called by users, but rather to be specialized with methods, when defining new Statice types. Each physical type must provide a method for this generic function. See the section "Defining a Variable-Format Physical Type", page 89.

See the section "Defining a Fixed-Format Physical Type", page 92.

After the handler itself, the first three arguments designate the first record. The second three arguments are also just like the first three arguments to designate the second record. For more information on the arguments: See the section "Arguments to Methods for Defining New Statice Types", page 99.

statice-type:record-equal *handler addressor-1 word-offset-1 n-words-or-bit-offset-1 addressor-2 word-offset-2 n-words-or-bit-offset-2* *Generic Function*

Methods are given two records, and must determine whether they are equal. Methods should return a true value if and only if the values stored in both records are not the null value and are equal to each other. Note that the rules for handling null values are not exactly analogous to those of the **statice-type:value-equal** method: if the value in either or both record is the null value, **statice-type:record-equal** must return **statice-type::nil**.

This generic function is not intended to be called by users, but rather to be specialized with methods, when defining new Statice types. Each physical type must provide a method for this generic function. See the section "Defining a Variable-Format Physical Type", page 89.

See the section "Defining a Fixed-Format Physical Type", page 92.

After the handler itself, the first three arguments designate the first record. The second three arguments are also just like the first three arguments to designate the second record. For more information on the arguments: See the section "Arguments to Methods for Defining New Statice Types", page 99.

statice:schema-name *schema* *Function*

Returns the name of the schema instance indicated by *schema*.

The *schema* argument can be either an instance of a real schema (returned by **statice:get-real-schema**) or of a template schema (returned by **statice:get-template-schema**).

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:schema-types *schema* *Function*

Returns a list of entity types of the given *schema*.

The *schema* argument can be either an instance of a real schema (returned by **statice:get-real-schema**) or of a template schema (returned by **statice:get-template-schema**).

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:set-attribute-value-array-portion *entity-handle attribute start end from from-start* *Function*

Writes from an array into a portion of an array-valued attribute.

entity-handle	An entity handle.
attribute	A symbol, which can be either the name of the attribute, or the name of its reader function. The value of this attribute can be a vector or a string.
start	The position in which to start writing into the array.
end	The position in which to stop writing into the array.
from	The array from which to read.
from-start	The position in the *from-array* where the reading should begin.

statice:set-attribute-value-to-null *entity-handle attribute* *Function*

Sets the value of an attribute to null; this function can choose the entity or attribute at run time.

entity-handle is an entity handle. *attribute* is a symbol, which can be either the name of the attribute, or the name of its reader function.

statice:set-attribute-value-to-null works with any attribute. In contrast, using **setf** with **statice:attribute-value** with **nil** as the value sets the attribute value to null only if **nil** is not a valid Lisp representation of some value of the type.

The following example asserts that it is not known whether Professor Smith is a visiting instructor. The value of **prof-smith** is an entity handle.

```
(set-attribute-value-to-null prof-smith 'instructor-visiting)
```

For more information:

See the section "The Statice Null Value", page 29.
See the section "Dynamic Statice Accessor Functions", page 101.

dbfs:set-buffer-replacement-parameters &key *(:page-pool-factor* **0.25***) (:page-pool-limit* **(* 1024 1024)***)* *Function*

Enables the user to limit the amount of virtual memory Statice will use as least-recently-used (LRU) buffer space. The more pages touched by concurrent transactions, the larger the buffer requirements are. The maximum number of words Statice will use as LRU buffer space is approximately equal to:

page-pool-factor * (*physical-memory-size* - *page-pool-limit*)

where *physical-memory-size* is the number of words of physical memory on the local machine. For example, on a 2MW machine using the default settings, Statice would be authorized to use up to 1/4 of a megaword of virtual memory to maintain its LRU buffer pool.

The best settings of these parameters depend on your particular application and hardware configuration. For a machine being used as a dedicated Statice server, you may wish to specify a *page-pool-factor* above 25%. For a machine being used for many applications, with only a moderate use of Statice, you may wish to specify a *page-pool-factor* below 25%.

statice:terminate-database *pathname* *Function*

Terminates the database stored in *pathname*, which is a database pathname.

Terminating a database is a way to undo the effects of opening a database. Note that, like opening a database, this does not have any effect on the persistent state of the database; it affects only the state within your own Lisp environment.

In the usual case, there is no need to terminate a database. Terminating a database allows entity handles associated with the database to be garbage collected. You can terminate and then open a database to inform the database that the schema name has changed (this is necessary when you change the package of a Statice program): See the section "Warning About Changing the Package of a Statice Program", page 58.

The primary reason to terminate a database is to enable you to deal with two exact copies of a single database. (You might have two exact copies if you use the Copy Database command or the backup dumper).

Every Statice entity has its own unique identity, represented internally by a numerical unique ID. Now, if you have databases A and B, and B is a copy of A, you could access two distinct entities (one in each database) that had

the same unique ID. Statice would get very confused, because the unique IDs would not really be unique. This cannot be allowed. Therefore, you cannot open two database at once if one of them is a copy of the other.

If A is open, and you try to open B, an error is signalled. The message is something like this:

```
The database X:>a exists and has the same
unique ID as the database in file X:>b
```

If you really want to open B, you must first terminate A. This is the purpose of **statice:terminate-database**.

When the above error is signalled, Statice provides a proceed handler that offers to terminate the database that's open. In the example, the proceed handler would offer to terminate X:>a.

statice:type-area-name *entity-type* *Function*

Returns the name of the area in which entities of the given *entity-type* are stored.

The *entity-type* argument is an entity type instance, such as one of the entity types in the list returned by **statice:schema-types** or the result of **statice:get-template-entity-type**.

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:type-attributes *entity-type* *Function*

Returns the names of the attributes of the given *entity-type*.

The *entity-type* argument is an entity type instance, such as one of the entity types in the list returned by **statice:schema-types** or the result of **statice:get-template-entity-type**.

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:type-multiple-indexes *entity-type* *Function*

Returns a list of multiple index instances of the given *entity-type*. If the entity type has no multiple indexes, this list is empty.

The *entity-type* argument is an entity type instance, such as one of the entity types in the list returned by **statice:schema-types** or the result of **statice:get-template-entity-type**.

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:type-name *entity-type* *Function*

Returns the symbol that is the name of the given *entity-type*.

The *entity-type* argument is an entity type instance, such as one of the entity types in the list returned by **statice:schema-types** or the result of **statice:get-template-entity-type**.

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:type-parent-names *entity-type* *Function*

Returns the names of the parent types of the given *entity-type*.

The *entity-type* argument is an entity type instance, such as one of the entity types in the list returned by **statice:schema-types** or the result of **statice:get-template-entity-type**.

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice:type-set-exists *entity-type* *Function*

Returns true if a set exists for the given *entity-type*; otherwise, returns **nil**.

The *entity-type* argument is an entity type instance, such as one of the entity types in the list returned by **statice:schema-types** or the result of **statice:get-template-entity-type**.

This function is not intended for use in Statice application programs, but rather in programs that need to examine all kinds of schemas, such as a browsing tool. See the section "Examining the Schema of a Statice Database", page 110.

statice-type:size-of-value *handler value* *Generic Function*

Methods for physical types should return the number of words of a record that would be used to represent this value. Statice uses this method to determine how much space must be allocated to store a value into a record.

This generic function is not intended to be called by users, but rather to be specialized with methods, when defining new Statice types. Each physical type must provide a method for this generic function. See the section "Defining a Variable-Format Physical Type", page 89.

See the section "Defining a Fixed-Format Physical Type", page 92.

For information on the arguments: See the section "Arguments to Methods for Defining New Statice Types", page 99.

statice-type:value-compare *handler value addressor word-offset n-words-or-bit-offset* *Generic Function*

Methods for physical types that are comparable receive a Lisp representation of a value and a record holding a value. They must return **:lessp** if the value stored in the record is less than the *value* argument. They return **:greaterp** if the record is greater than the *value*, and **:equal** if the two values are equal. If the value in the record is the null value, methods must return the null value: **statice-type:*null-value***. The *value* passed as an argument is never the null value.

The advantage of having an explicit **statice-type:value-compare** method, instead of using **statice-type:read-value** and comparing the the Lisp representations, is to avoid allocating Lisp storage (consing) to build a Lisp rational number, resulting in greater efficiency.

This generic function is not intended to be called by users, but rather to be specialized with methods, when defining new Statice types. See the section "Defining a Variable-Format Physical Type", page 89.

See the section "Defining a Fixed-Format Physical Type", page 92.

For information on the arguments: See the section "Arguments to Methods for Defining New Statice Types", page 99.

statice-type:value-equal *handler value addressor word-offset n-words-or-bit-offset* *Generic Function*

Methods for physical types should return true if the value in the record is considered equal to the *value* argument. The null value must be considered equal only to the null value, and no other value.

This generic function is not intended to be called by users, but rather to be specialized with methods, when defining new Statice types. Each physical type must provide a method for this generic function. See the section "Defining a Variable-Format Physical Type", page 89.

See the section "Defining a Fixed-Format Physical Type", page 92.

For information on the arguments: See the section "Arguments to Methods for Defining New Statice Types", page 99.

statice:view-entity *stream pathname entity-handle* &optional *(setf-function* **#'statice::make-browser-attribute-value-setf***) values* *Function*

Displays an arbitrary entity in a window.

stream	The window on which to view the entity.
pathname	Specifies the database which the entity handle is from.
entity-handle	The entity to be viewed.
setf-function	Should be a function of two arguments, an entity-handle and the name of an attribute, which returns a list suitable for **setf**'ing.

Here's an example from the university database. Suppose the value of ***university-pathname*** is the pathname of the university database, and that the value of **j** is the entity handle for the student named Joe.

```
(view-entity *standard-output* *university-pathname* j)

#<STUDENT Joe 504260721>  (type STUDENT)
NAME:               Joe
ID-NUMBER:          827
DEPT:               (Null value)
COURSES:            (Entity-Handle of COURSE 24/28
                     Entity-Handle of COURSE 24/25)
SHIRTS:             (Entity-Handle of SHIRT 25/11)
```

The default *setf-function*, **#'statice::make-browser-attribute-value-setf** is defined as:

```
(defun make-browser-attribute-value-setf (entity-handle fname)
  `(browser-attribute-value ,entity-handle ',fname))
```

This function is called to create the **:form** argument to **present** for the attribute values. If `c-m-Right` is clicked on (i.e. "Modify this structure slot"), the user interface management system does a **setf** operation, using the **:form** argument and the new value that was entered. By default, the Browser uses **#'statice::make-browser-attribute-value-setf** as its **:form** function, so that when a attribute slot is modified, the following form is evaluated:

```
(setf (browser-attribute-value <entity-handle>
          'name-of-attribute-being-modified)
      new-value-entered)
```

browser-attribute-value is not defined as a function, but **(setf browser-attribute-value)** is, and may be used to **setf** a single attribute value in an entity handle. Note that it performs its own transaction.

statice:with-cluster *(cluster-spec)* &body *body* *Special Form*

Defines the "current cluster" for the dynamic extent of the *body*. When you make a new entity within the dynamic extent of *body*, the entity is created inside the current cluster. This overrides the declarative interface and any area specifications. The *cluster-spec* is evaluated at run time, so it may be a variable.

cluster-spec is one of these:

:new	Places all entities that are created in the dynamic scope of body in a new cluster. That is, the first entity to be created in the **statice:with-cluster** form is placed in its own cluster, and the rest of the entities are placed in that same cluster.
:none	Turns off clustering.
entity-handle	Places all new entities created in the dynamic scope of *body* in an existing cluster—that of the *entity-handle*. If

entity-handle is not a clustered entity, then no clustering takes place, and the entities are allocated wherever there is free space on non-clustered pages

See the section "Clustering Technique for Statice Databases", page 133.

statice:with-current-database *(database)* &body *body* *Special Form*

Makes the value of the form *database* be the current database, for the (dynamic) extent of its *body*. The value must be a database object.

Along with **statice:open-database**, **statice:with-current-database** can be considered a primitive underlying **statice:with-database**. In some cases, users can use these primitives to achieve greater speed, or to deal with more than one database (such as when copying data from one to another). For details: See the section "Opening and Terminating Databases", page 72.

statice:with-database *(variable pathname)* &body *body* *Special Form*

Any use of a Statice database must be surrounded by a **statice:with-database** form. **statice:with-database** does the following:

1. Determines which database should be opened, based on the *pathname*.
2. Opens the database, unless it is already open.
3. Binds the specified *variable* to the database instance, during the execution of the *body*. Database instances are used as arguments to various Statice functions. They are needed only by programs that refer to two different databases at the same time.
4. Makes this the current database, throughout the dynamic scope of the *body*.
5. Executes the *body*.

For example:

```
(defun make-new-account (new-name new-balance)
  (with-database (db *bank-pathname*)
    (with-transaction ()
      (make-account :name new-name :balance new-balance))))
```

See the section "Accessing a Statice Database", page 9.

statice:with-transaction *(*&key *(:automatic-retry* **'dbfs:restartable-transaction-abort***))* &body *body* *Special Form*

Every operation that examines or modifies a database must be done inside (dynamically) a **statice:with-transaction** form.

The dynamic extent of the **statice:with-transaction** form delimits a *transaction*. A transaction is group of operations on a database that is guaranteed to be *atomic*, *isolated*, and *persistent*.

For example:

```
(defun transfer-between-accounts (from-name to-name amount)
  (with-database (db *bank-pathname*)
    (with-transaction ()
      (decf (account-balance (account-named from-name)) amount)
      (incf (account-balance (account-named to-name)) amount)
      (when (minusp (account-balance (account-named from-name)))
        (error "Insufficient funds in ~A's account" from-name)))))
```

The keyword option is:

:automatic-retry The value is the name of a flavor. Whenever a transaction is aborted, some error is signalled. If the error flavor is a subtype of the specified flavor, the transaction automatically restarts; otherwise, the error is signalled in the dynamic context of **statice:with-transaction** (that is, a **throw** is done to the point of the **statice:with-transaction** form, and then the error is signalled). By using this keyword, you can control which errors will cause automatic restarts and which will not. This keyword is rarely used.

See the section "Introduction to Statice Transactions", page 10.

statice-storage:write-multiple-record-word *record-addressor start-index end-index new-value* *Generic Function*

Writes a contiguous subsequence of words from an array into the record specified by *record-addressor*, starting at *start-index* and ending at *end-index*. *new-value* must be an array of signed 32-bit integers. The array is made and returned on the data stack, so the caller is required to provide a **sys:with-data-stack** special form around the dynamic extent of the use of the array.

This generic function is used only in the methods for defining physical types: See the section "Defining Physical Types", page 88.

statice-storage:write-record-word *record-addressor index new-value* &key *:buffer-p* *Generic Function*

Writes *new-value* into the word specified by *index* of the record specified by *record-addressor*. *new-value* must be a signed 32-bit integer. *record-addressor* must be writable.

This generic function is used only in the methods for defining physical types: See the section "Defining Physical Types", page 88.

statice-type:write-value *handler value addressor word-offset n-words-or-bit-offset* *Generic Function*

Methods for physical types should write the *value* into the portion of the record, or write an indication that the value is null.

This generic function is not intended to be called by users, but rather to be specialized with methods, when defining new Statice types. Each physical type must provide a method for this generic function. See the section "Defining a Variable-Format Physical Type", page 89.

See the section "Defining a Fixed-Format Physical Type", page 92.

9. Dictionary of Statice Error Flavors

statice:database-deleted *Flavor*

A transaction has attempted to use a database that has been deleted.

Function *Value*

statice:database-deleted-pathname
The pathname of the database.

statice:database-terminated *Flavor*

A transaction has attempted to use a database that has been terminated.

See the section "Opening and Terminating Databases", page 72. See the function **statice:terminate-database**, page 216.

statice:entity-handle-deleted *Flavor*

There was an attempt to use an entity handle whose entity has been deleted.

Function *Value*

statice:entity-handle-deleted-entity-handle
The deleted entity handle.

See the section "Deleting Entities". See the function **statice:delete-entity**, page 196.

statice:entity-handle-not-committed *Flavor*

There was an attempt to use an entity handle that was created in a transaction that was aborted. This error probably indicates that you passed an entity handle out of a transaction which aborted, and then tried to use the entity handle.

Function *Value*

statice:entity-handle-not-committed-entity-handle
The entity handle.

statice:not-inside-transaction *Flavor*

There was an attempt to use Statice while not inside any transaction.

See the section "Introduction to Statice Transactions", page 10. See the special form **statice:with-transaction**, page 221.

statice:entity-not-found-in-set *Flavor*

An attempt was made to delete an item from a set (using **statice:delete-from-set** or **statice:delete-from-set***), but the item was not a member of the set.

Function *Value*

statice:entity-not-found-in-set-entity-handle
The value that you were trying to remove from the set.

statice:entity-not-found-in-set-set-of-entities
A list of the of values in the set.

This flavor is built on **statice:function-error**. See the special form **statice:delete-from-set**, page 197. See the function **statice:delete-from-set***, page 197.

statice:function-error *Flavor*

All errors at the "function level" of Statice are built on this flavor.

This flavor is built on **error**.

statice:function-uniqueness-violation *Flavor*

There was an attempt to violate an attribute's **:unique t** constraint, by making a new entity or by changing the attribute value of an existing entity.

Function *Value*

statice:function-uniqueness-violation-type
The type instance.

statice:function-uniqueness-violation-function
The attribute instance.

This flavor is built on **statice:uniqueness-violation**. See the section "One-to-One, Many-to-One, and Other Relationships", page 24. See the special form **statice:define-entity-type**, page 187.

statice:index-uniqueness-violation *Flavor*

There was an attempt to violate the **:unique t** constraint of a multiple index.

Function *Value*

statice:index-uniqueness-violation-type
The type instance.

statice::index-uniqueness-violation-functions
A list of attribute instances, specifying the multiple index.

This flavor is built on **statice:uniqueness-violation**. See the section "Introduction to Multiple Indexes", page 51. See the special form **statice:define-entity-type**, page 187.

statice:no-current-database *Flavor*

There was an attempt to use Statice, but there was no current database. Possibly the program is missing **statice:with-database** or **statice:with-current-database**.

This flavor is built on **statice:function-error**. See the special form **statice:with-database**, page 221.

See the special form **statice:with-current-database**, page 221.

statice:no-entity-type-named *Flavor*

An attempt was made to try to use an entity type that doesn't exist in the current database.

This flavor is built on **statice:function-error**.

statice:no-function-named *Flavor*

An attempt was made to try to use a function whose attribute doesn't exist in the current database.

This flavor is built on **statice:function-error**.

statice:schema-not-loaded *Flavor*

A database was opened that had a real schema for which the corresponding template schema was not loaded into the Lisp world.

Function *Value*

statice:schema-not-loaded-pathname
Pathname of the database.

statice:schema-not-loaded-schema-description-name
The name of the template schema that Statice expected to find.

This flavor is built on **statice:function-error**. See the section "Template Schemas and Real Schemas", page 110. See the section "Warning About Changing the Package of a Statice Program", page 58.

statice:uniqueness-violation *Flavor*

All errors that violate **:unique t** constraints are built on this flavor.

This flavor is built on **statice:function-error**.

statice:value-not-a-set *Flavor*

The value given as the new value for a set-valued attribute was not a list. This error can be signalled by attribute writer functions or entity creation functions. The error message is "Expected a list of values".

This flavor is built on **statice:function-error**.

See the section "Set-Valued Attributes", page 22.

statice:wrong-type-entity *Flavor*

Signalled by many of the dynamic functions, such as **statice:add-to-set*** and **statice:attribute-value**, to indicate that an entity argument was of the wrong type. You can call **statice:wrong-type-entity-entity-handle** to get the entity which is of the wrong type, and **statice:wrong-type-entity-expected-type** to find out what type was expected.

Function *Value*

statice:wrong-type-entity-entity-handle
Entity of the wrong type

statice:wrong-type-entity-expected-type
What type was expected.

This flavor is built on **statice:function-error**. See the function **statice:add-to-set***, page 182. See the function **statice:attribute-value**, page 185.

Index