Sun StorageTek 5800 System Client API Reference Manual

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 820–4796
June 2008

080616@20490

Contents

Tables

Preface

The *Sun StorageTek 5800 System Client API Reference Manual* is written for programmers and application developers who develop custom applications for the Sun StorageTek™ 5800 System. This document, along with the *Sun StorageTek 5800 SystemSDK Reference Manual*, provides the information that you need to develop custom applications for the 5800 system.

How This Book Is Organized

- Chapter 1, "Sun StorageTek 5800 System Client API," provides a summary of the changes for the Sun StorageTek 5800 System 1.1 release, and overviews of the client APIs and query language.
- Chapter 2, "Sun StorageTek 5800 System Java Client API," provides detailed information on the Sun StorageTek 5800 System Java client API.
- Chapter 3, "Sun StorageTek 5800 System C Client API," provides detailed information on the Sun StorageTek 5800 System C client API.
- Chapter 4, "Sun StorageTek 5800 System Query Language," provides detailed information on the Sun StorageTek 5800 System query language.
- Chapter 5, "Programming Considerations and Best Practices," provides programming considerations and best practices that can help you create efficient 5800 system applications.

Related Books

- *Sun StorageTek 5800 System Regulatory and Safety Compliance Manual*, part number 819–3809
- *Sun StorageTek 5800 System Site Preparation Guide*, part number 820–1635
- *Sun StorageTek 5800 System Administration Guide*, part number 820–4118
- *Sun StorageTek 5800 System SDK Reference Manual*, part number 820–4797
- *Sun StorageTek 5800 System 1.1.1 Release Notes*, part number 820–4120

Related Third-Party Web Site References

Third-party URLs are referenced in this document and provide additional, related information.

Note – Sun is not responsible for the availability of third-party web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

Documentation, Support, and Training

The Sun web site provides information about the following additional resources:

- Documentation (`http://www.sun.com/documentation/`)
- Support (`http://www.sun.com/support/`)
- Training (`http://www.sun.com/training/`)

Typographic Conventions

The following table describes the typographic conventions that are used in this book.

TABLE P–1 Typographic Conventions

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`** `Password:`
aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is `rm` *filename.*

TABLE P–1 Typographic Conventions *(Continued)*

Typeface	Meaning	Example
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file. **Note:** Some emphasized items appear bold online.

Shell Prompts in Command Examples

The following table shows the default UNIX® system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell	`machine_name%`
C shell for superuser	`machine_name#`
Bourne shell and Korn shell	`$`
Bourne shell and Korn shell for superuser	`#`

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. You can submit your comments by clicking the Feedback link on the http://docs.sun.com web site.

Please include the title and part number of your document with your feedback:

Sun StorageTek 5800 System Client API Reference Manual, part number 820-4796

◆ ◆ ◆ CHAPTER 1

Sun StorageTek 5800 System Client API

The Sun™ StorageTek™ 5800 system client API provides programmatic access to a 5800 system server to store, retrieve, query, and delete object data and metadata. Synchronous versions are provided in C and Java™ languages. A future release will implement a non-blocking C API for use with POSIX operations.

This chapter provides a summary of the changes for the Sun StorageTek 5800 System 1.1 release, and overviews of the client APIs and query language.

The following topics are discussed:

Changes in Version 1.1

The following general changes have been made in Version 1.1.

- Handling is added for storing, retrieving, and querying the following metadata types:
  - char — for Latin 1 character set
  - string — for Unicode character set
  - binary
  - date
  - time
  - timestamp
- Query and queryplus are merged.
- Prepared statements (pstmts) are introduced to handle the values of queries that cannot be placed inline, and a new query is introduced to handle them.
- The handling of strings that are longer than the string length of the associated field has changed.

In 5800 system version 1.1, an attempt to store a value that is longer than the associated field generates an immediate error.

5800 System Overview

This section provides an overviews of the 5800 system, the 5800 system history, and a summaries of the key points of the 5800 system usage model.

The following topics are discussed:

5800 System Summary

The 5800 system is an object-based storage archive appliance for fixed-content data and metadata. The 5800 system is designed from the ground up to be reliable, affordable, and scalable, and to integrate data storage with intelligent data retrieval. It is designed to store huge amounts of data for decades at a time. At that scale, issues of how and where the data is stored — and how that changes over time — can be quite cumbersome. The 5800 system usage model is designed to manage those issues for you, so that your application can deal with just the data.

A custom Application Programming Interface (the 5800 Client API) is provided so that your applications can take advantage of all the features in the 5800 system usage model. The API provides the following capabilities:

- Store a new object into the archive (`storeObject`)
- Associate a new metadata record with stored object data (`storeMetadata`)
- Retrieve the data from an object that was previously stored (`retrieveData`)
- Retrieve the metadata from an object that was previously stored (`retrieveMetadata`)
- Delete an object (`delete`)
- Query for matching objects given a query expression of specific object characteristics (`query`)

The 5800 system API Release 1.1 provides two APIs:

- The Java API is described in Chapter 2, "Sun StorageTek 5800 System Java Client API"
- The C API is described in Chapter 3, "Sun StorageTek 5800 System C Client API"

This chapter provides a summary of key points of the 5800 system usage model that are useful for understanding either API.

In the following sections, the terms from the Java API are used as an aid to exposition. In all cases, a simple equivalent using the C API is available.

- Chapter 4, "Sun StorageTek 5800 System Query Language," provides a detailed description of query capabilties and query syntax.
- Chapter 5, "Programming Considerations and Best Practices," provides programming considerations and best practices that can help you create efficient 5800 system applications.

The 5800 System and Honeycomb

The original code name for the project that grew into the 5800 system was *Project Honeycomb*. The Honeycomb name lives on as the name of an Open Solaris community that is bringing the Honeycomb software stack into the world of Open Source. The first realization of the Honeycomb storage model as a real product is the 5800 system as described in this guide and related guides.

As a model for programmable storage systems, however, the Honeycomb API has a much broader reach than just the 5800 system. The programming model is designed to scale both up and down to any storage archive system that needs to abstract and separate the issues of how data is stored from how it is used. In recognition of both the past and the future, the string "`honeycomb`" and the initials "`hc`" still live on in certain aspects of the API described in this guide. When the 5800 system API is used in contexts outside of the 5800 system, the API is referred to as the *Project Honeycomb API*.

The 5800 System Data Model

The 5800 system stores two types of data: arbitrary object data and structured metadata records. Every metadata record is associated with exactly one data object. Every data object has at least one metadata record. A unique object identifier (OID) is returned when a metadata record is stored. This OID can later be used to retrieve the metadata record or its data object. In addition, metadata records can be retrieved by a query:

OID $\leftrightarrow$ *Metadata Record* $\rightarrow$ *Underlying Object Data*

There are two types of metadata, system metadata and user metadata. You cannot override the names and types of system metadata.

Each object in the 5800 system archive consists of some arbitrary bytes of data together with associated metadata that describes the data. Once an object is stored, it is immutable. The 5800 system programming model does not allow the data or the metadata associated with an object to be changed once the object has been stored, in other words the system is a Write-Once

Read-Multiple (WORM) archive. Each object corresponds to a single stream of data and a single set of metadata; there are no “grouped objects” or “compound objects” other than by application convention.

Each object corresponds to a single stream of data and a single set of metadata. There are no “grouped objects” or “compound objects” other than by application convention. Similarly, there are no “links” or “associations“ from one object to another. The customer application is shielded from all details of how or where the object is stored. Internally, the actual location of an object might change over time, or several objects might even share the same underlying storage. The customer application can retrieve the object without knowing these details.

A stream of data is stored in the object archive using `storeObject`. Once stored, each such object is associated with an *object identifier* or *objectid* (OID). The `storeObject` operation takes both a stream of data and an optional set of user metadata information and returns an OID. The OID can be remembered outside of the 5800 system and may later be used to retrieve the data associated with that object using the `retrieveObject` operation.

Every object has metadata whether or not user metadata was supplied at the time of the store. For example each object has system metadata that is system assigned and can never be modified by the user. The OID is associated with the metadata record that represents this object as a whole; the metadata record is then associated with the underlying data:

OID ↔ Metadata Record → Underlying Object Data

The `retrieveObject` operation takes an OID as input and returns a stream of bytes as output that are identical to the bytes stored during the `storeObject` operation. Both the `storeObject` and `retrieveObject` operations handle the data in a streaming manner. Not all of the data need be present in client memory or in server memory at the same time, which is a crucial point for working with large objects.

For the 5800 system Release 1.1, data sizes up to 400 GBytes are tested and supported. Using sizes even smaller than this may be appropriate as a best practice. For more information, see Chapter 5, “Programming Considerations and Best Practices.”

From within a customer application, the storing of an object into the archive is an all-or-nothing event. Either the object is stored or it is not; there are no partial stores. If a store operation is interrupted, the entire `storeObject` call fails. Once an OID is returned to the customer application, the object is known to be durable. In the event of an outage that causes some data loss, the system should be no more likely to lose a newly stored object than any other object. There is no way to tie together two different store operations so that both either succeed together or fail together.

Note – A stored object may or may not immediately be queryable. For more information, see “The 5800 System Query Integrity Model” on page 21.

The 5800 System Metadata Model

Metadata means "data about the data"; it describes the data and helps to determine how the data should be interpreted. In addition, metadata can be used to facilitate querying the 5800 system for objects that match a particular set of search criteria.

For the 5800 system, the supported metadata option is in the form of name-value fields stored with each object. The set of possible fields is defined in the metadata schema. Setting up a metadata schema is an important system administration task that is described in the *5800 System Administration Guide*, and is analogous to the process of database design that goes into creating a data management application. The metadata schema determines what field names, types, and lengths may be used with the metadata stored with each object. In addition, the layout of fields into tables within the schema, together with the definition of views that speed certain searches, determine which kinds of queries about that metadata will be both possible and effective. As such, the metadata schema should match the characteristics of the expected range of applications that will deal with the stored data. The underlying software is designed to support multiple different kinds of metadata to aid in searching. For example, eventually there might be a specialized index to facilitate full-text search within the data objects. This document describes only the API for dealing with the name-value metadata type.

Fields in the schema can be either *queryable* or *non-queryable*. The values for non-queryable fields may be retrieved later but may not be used in queries. The 5800 system supports only single-valued fields. Each object can have only a single name-value pair of a given name. There is no built-in support for multiple-valued fields, such as a list of authors of a book in the form of multiple fields named 'author'.

Each data object is associated with a set of name-value pairs at the time the object is stored. Some metadata (system metadata) is assigned by the5800 system as each object is stored. For example, each object contains an "object creation time" (`system.object_ctime`) and an OID (`system.object_id`), both of which are assigned by the system at the time an object is created. Some metadata (the computed metadata) is implicit in the stored data, and is made explicit at the time of the object store. For example, the system exposes the object data length as a metadata field (`system.object_size`). In addition, the 5800 system computes a Secure Hash Algorithm (SHA1) hash of the stored data as the data is stored and stores the hash as a metadata field (`system.object_hash`). There is also an associated field (`system.object_hash_alg`) to specify which hash algorithm was used in computing the `system.object_hash`. It is currently always set to "`sha1`."

Finally, some metadata (the user metadata) is supplied by the customer application in the API call at the time an object is stored. Each store operation is allowed to include a `NameValueRecord` that indicates a set of name-value pairs to be associated with the data object as metadata. Each name in the name-value record must match a field name from the metadata schema; in addition, the data value supplied for each field must match the type and length for the field as specified in the schema. If the names or values supplied for the user metadata do not match the active schema, then an exception is generated and the object is not stored.

The metadata associated with an object is immutable. There is no operation to modify the metadata associated with an object after the object has been stored. Instead, the `storeMetadata` operation can be used to create a completely new object by associating new user metadata with the underlying data and system-metadata of an existing object. The `storeMetadata` operation does not merge the new metadata in with the metadata from the original OID; instead, the `storeMetadata` operation creates a new metadata record pointing to the same data object. To accomplish a merge of new field values into existing metadata, the customer application must manually retrieve the existing metadata from the original object, perform the merge into a single `NameValueRecord` on the client side, and then call `storeMetadata` to create a new object with the merged metadata.

When creating a new object using `storeMetadata`, a new `system.object_id` and new `system.object_ctime` are generated, to indicate that a new object has been created. The metadata computed from the object data itself (`system.object_length`, `system.object_hash_alg`, and `system.object_hash`) does not change. Both the `storeObject` and the `storeMetadata` operations return a `SystemRecord` value that includes all of the system-assigned fields.

While retrieving the OID is the most common use of the `SystemRecord`, the other `system` fields can also be helpful. For example, the customer application might use the `system.object_length`, the `system.object_hash_alg` and the `system.object_hash` fields to verify that the data as stored matches the data as present in the customer application. If a hash independently computed on the client matches the hash stored on the 5800 system, then the data store has been validated.

The metadata values associated with an object can be retrieved using the `retrieveMetadata` operation. The `retrieveMetadata` operation takes an OID as input, and returns the entire set of user, system, and system-computed metadata. The retrieved metadata is in the form of a `NameValueRecord` that contains the value of each field as originally stored. The system fields occur using their field names, for example. the field `system.object_ctime` contains the object creation time. There is no operation to retrieve just a single field or a subset of fields by supplying a list of field names. The `retrieveMetadata` operation retrieves the values of both `queryable` and `non-queryable` fields.

The 5800 System Query Model

One of the primary methods for retrieving data is to specify the characteristics of the desired data and then let the system find it for you. In the 5800 system, a query expression specifies a set of conditions on metadata field values. The system then returns a list of all the objects whose metadata values match the query conditions. Each object is considered individually without reference to any other objects. There are no queries that compare fields in one object with fields in a different object.

Query expressions can use much of the power of Structured Query Language (SQL). Each query expression combines SQL functions and operators, field names from the metadata schema, and literal values. There are no query expressions that select objects based on the data stored in the object itself; all queries apply only to the metadata fields associated with the object. Only queryable fields can be used in query expressions. For an object to show up in a query result set, the object must have a value for each of the fields mentioned in the query; in other words, there is an implicit INNER JOIN between the fields in the query.

A query may optionally specify that the result set should include not just the OID of each matching object, but also the values from a set of selected fields of each matched object . The value retrieved by Query With Select for some field may be a canonical equivalent of the value originally stored in that field. For example, values in numeric fields may have been converted to standard numeric format. Trailing spaces at the end of string fields will have been truncated (The value that is returned will be some value that would match the original data as stored, in the SQL sense.) To be included in the result set, an object must include values for all queried fields and all selected fields. In other words, there is an implicit INNER JOIN between all the fields in the query and in the select list.

There are significant limitations on which queries may be executed efficiently, or at all. See Chapter 4, "Sun StorageTek 5800 System Query Language," and Chapter 5, "Programming Considerations and Best Practices"s for details of these limitations.

There are no ordering guarantees between queries and store operations that are proceeding at the same time. If an object is added to the 5800 system while a query is being performed, and the object matches the query, then the object may or may not show up in the query result set.

For a detailed description of query syntax and query semantics, including a description of exactly what it means for an object to match a query, see Chapter 4, "Sun StorageTek 5800 System Query Language."

The 5800 System Query Integrity Model

The result set of any query will only return results that match the query. But will it return ALL the matching results? That is the concept of query completeness, referred to here as *query integrity*. 100% query integrity for a result set is defined as a state in which the result set contains all the objects in the 5800 system that match that particular query. The 5800 system is not always in a state of 100% query integrity. Various system events can induce a state in which the set of objects that are available for query is smaller than the total set of objects stored in the archive. Each query result set supports an operation (`isQueryComplete`) whereby the customer application can ask, once all the results from the query result set have been processed, whether that set of results constitutes a complete set.

Note – The format of records as stored in the reliable and scalable object archive is not suitable for fast query. To enable searching, the queryable fields from the metadata are indexed in a query engine that can provide fast and flexible query services. The query engine is basically an SQL database. This is why the 5800 system's query language can borrow so heavily from SQL. At various times, the data as indexed in the query engine can get out of date compared to what is stored in the archive. When this happens, query result sets are not known to be complete until the contents of the query engine can be brought back up to date with the actual contents of the archive again.

The 5800 system concept of query integrity as actually implemented is somewhat looser than that of 100% query integrity. Even if a query result set indicates the result set is complete, the 5800 system allows certain objects, known as *store index exceptions*, to be missing from the query result set, as long as those exceptions were communicated to the customer application at the time the object was stored.

A store index exception is an object for which the original store of the object into the archive succeeded, but at least some part of the insert into the query engine (database) did not succeed. The object may or may not show up in all of the queries that it matches. A store index exception is communicated to the customer application at the time of store by means of a method `SystemRecord.isIndexed`. A value of false from `isIndexed` means that the object is not immediately available for query.

A store index exception is said to be resolved when the object becomes available for query. The `checkIndexed` method can be used to attempt to resolve a store index exception under program control. The `checkIndexed` operation checks if the object has been added to the query engine, and attempts to insert it if the object has not been added. If the insert into the query engine succeeds, the object is thereby restored to full queryability.

All store index exceptions will also eventually be resolved automatically by ongoing system healing. Each query result set also exports a method `getQueryIntegrityTime` that can be used to get detailed status on which store index exceptions might still be unresolved. The query integrity time is a time such that all store index exceptions from before that time have been resolved. There is an “ideal” query integrity time, which is the time of the oldest still-unresolved store index exception: an ideal implementation when asked for the query integrity time would always report this ideal value. In actual implementation, the reported query integrity time might be hours or even days earlier than the ideal query integrity time, depending on how far the ongoing system healing has progressed.

Deleting Objects from the 5800 System

The 5800 system client API exports an operation to delete a specific object as specified by its OID. Once a delete operation completes normally, subsequent attempts to retrieve that object will fail with an exception. In addition, the object will stop showing up in query result sets that

match the original object metadata. There are no transactional guarantees regarding ordering of queries and delete operations that are occurring at the same time. If an object is being deleted at the same time that a query that matches that object is being performed, then that object may or may not show up in the query result set, with no guarantee either way.

Note – When all objects that share an underlying block of data storage been deleted, the underlying block of data storage will itself be scavenged and returned to the supply of free disk space. But all details of how objects are stored, and how and whether they ever share data — or ever are scavenged — are outside of the scope of this API.

Delete operations are all-or-nothing,with some caveats. Specifically, if a delete operation fails with an error, it is possible that the object is not fully deleted but is temporarily not queryable. Such an object is in an analogous state to a store index exception (see "The 5800 System Query Integrity Model" on page 21). The queryability of such an object will eventually be resolved by automatic system healing. In addition, the queryability of such an object can be resolved under program control by using the `checkIndexed` method. Alternatively, the customer application may choose to re-execute the delete operation until it succeeds, or until it fails with an error that indicates the object is already deleted.

◆ ◆ ◆ CHAPTER 2

Sun StorageTek 5800 System Java Client API

This chapter provides information on the 5800 system Java client API.

The following topics are discussed:

Note – You can find detailed information on the 5800 system Java client API in the Javadocs, which are located in the `java/doc/htdocs` directory.

Overview of the 5800 System Java Client API

This section provides an overview of the 5800 system Java client API. The following topics are discussed:

Client Library

The 5800 system Java client library provides a simple way to communicate with 5800 system clusters. It provides programmatic access to the 5800 system network protocol, which operates over HTTP, enabling you to store, retrieve, query, and delete object data and metadata.

The 5800 system Java client library provides a platform-independent mechanism to upload data and metadata to a 5800 system, and to retrieve and query the data and metadata. The Java client library works with any implementation of J2SE™ platform 4.0 or later with HTTP connectivity to the 5800 system cluster. Access is designed to be high-level and easy to use. Most operations are accomplished in a single (synchronous) function call.

Interfaces

The Java client API interacts with the 5800 system server entirely through an HTTP protocol. The HTTP communication layer uses the Apache Commons HTTP client.

Object data is streamed through the Java client library opaquely and a well-defined data hash is returned for verification purposes. Metadata is added or retrieved with typed accessors. The stored representation of metadata on the 5800 system server is not exposed to the user, and no hash is returned when metadata is stored.

The 5800 system Java client library provides the `NameValueObjectArchive` class as an application access layer, which should be appropriate for most applications. In addition, an advanced interface provides a mechanism to customize the 5800 system and to serve as a toolkit to build new applications.

Note – The advanced toolkit is not described in this document. If you are interested in pursuing advanced applications, contact your 5800 system Sales Representative.

Retrying Operations

Calls to the Java API should be wrapped with retry logic so that their applications are resilient to transient failures that may be experienced when a node or switch fails while servicing an operation.

Requests that fail on recoverable HTTP errors are automatically retried once. A typical recoverable error occurs when the 5800 system HTTP server times out a connection that the client then tries to reuse (the client maintains a collection pool). This results in a connection failure at request time. Because this is a recoverable error, it is retried and the retry typically succeeds.

Performance and Scalability

Starting the Java Virtual Machine (JVM) incurs a performance penalty, but once the JVM is running, you can use the client object archive repeatedly and from multiple threads. I/O is synchronous (blocking). HTTP connections are pooled for performance. You should instantiate one instance of the `NameValueObjectArchive` per 5800 system server and use it for all access to that server until exit.

Updating Client View of the Schema

In the Java client API, the schema is fetched when the `NameValueObjectArchive` class is instantiated. If the schema has changed, the client application needs to create a new `NameValueArchive`. A local copy of the schema is used for some metadata operations.

Java Client Application Deployment

Java applications using the 5800 system Java API reference the `honeycomb-client.jar` library. You must include this library in your `classpath` when running your application. The 5800 system Java API was designed to run on Java v1.4, so you need to run your client applications with a Java environment of v1.4 or greater.

Java API

The 5800 system Java client library provides a simple way of communicating with 5800 system clusters. It provides programatic access to the 5800 system network protocol, which operates over HTTP. You can implement most applications using a handful of these classes, but access to "expert" features is also included.

This section discusses the following topics:

Java API Packages

The Java API is implemented in two packages:

- `com.sun.honeycomb.client`

  Provides the base classes required to interact with a 5800 system cluster.

- `com.sun.honeycomb.common`

  Contains classes for server-side exceptions.

Java API Documentation

The Java API documentation (Javadoc) is located in the SDK `java/doc/htdocs` directory, and can be accessed using a browser.

Basic Concepts

The root of the 5800 system Java client API is the `NameValueObjectArchive` class, which represents a connection to a single 5800 system server. All operations are initiated by invoking methods on a `NameValueObjectArchive` instance after initializing it with the address of a cluster. The fact that a cluster of machines, rather than a single server, is handling the requests is transparent to the application programmer.

A `NameValueObjectArchive` uses instances of the `ObjectIdentifier` class to uniquely identify stored data objects. That is, there is a one-to-one correspondence between instances of `ObjectIdentifer` and 5800 system metadata objects.

Note – There is potentially a many-to-one relationship between metadata and data objects.

When using `NameValueObjectArchive`, all metadata queries are executed against a 5800 system server's user-configurable index of name-value pair lists. This class also ensures that a metadata entry is created for every data object stored, even if no metadata is provided at store time.

An instance of the `NameValueObjectArchive` class functions as a proxy for the 5800 system server. Instantiation incurs some overhead in establishing communication, so reusing a single instance is the recommended practice. Multithreading is supported with the same instance.

`NameValueObjectArchive` also allows all metadata operations to be performed in terms of two classes that represent metadata records: `SystemRecord` and `NameValueRecord`. These classes represent 5800 system metadata entries. When using `NameValueObjectArchive`, every stored data object has a corresponding `NameValueRecord` that contains the extended attributes stored with that data object, and each `NameValueRecord` has a reference to its `SystemRecord`, which contains built-in system attributes such as data object size and creation time. In this model, all instances of `ObjectIdentifer` returned from store operations and metadata queries correspond directly to instances of `NameValueRecord`.

The results of a 5800 system metadata query are returned using instances of the `QueryResultSet` class, which the application can step through to retrieve metadata or identifiers. This class manages the details of fetching one batch of results after another.

Key Classes

This section provides an overview of the following key classes in the 5800 system Java client API. For more information on using the following classes, see "Basic Concepts" on page 28. Also see the Javadoc provided with the 5800 system SDK.

- "`NameValueObjectArchive`" on page 29
- "`NameValueSchema`" on page 29

For more information on using these classes, see "Basic Concepts" on page 28.

NameValueObjectArchive

The NameValueObjectArchive class is the main entry point into the 5800 system. Each instance of NameValueObjectArchive provides access to a specific 5800 system server, functioning as a proxy object on which operations can be performed. Multiple simultaneous operations can be accomplished in separate threads on the same NameValueObjectArchive instance. Communication with the 5800 system server is entirely by means of HTTP requests. A pool of HTTP connections is maintained for efficiency.

A NameValueObjectArchive instance enables you to store, retrieve, query and delete object data and associated metadata records. Metadata is associated with an object in a set of name-value pairs (see "NameValueRecord" on page 30). Metadata records can be used to associate application-specific information with the raw data, such as name, mime type, or purge date. Metadata records consist of structured data that can be queried. Object data is opaque to the 5800 system.

A NameValueObjectArchive instance always ensures that a metadata record is created on the 5800 system server for each newly stored object, even if no metadata is provided with the store. This enables a model of programming where every stored data object is accessed by name-value metadata records (for example, for examining results from queries or performing delete operations). Object data is never deleted directly; it is deleted when its last referencing metadata record is deleted.

For additional information, see "NameValueObjectArchive Application Access" on page 30.

NameValueSchema

An instance of NameValueSchema represents information about the name-value metadata that the 5800 system system uses to index data. This instance can be used to enumerate the fields available in the schema as attributes. Each attribute has a name and a type.

See the *Sun StorageTek 5800 System Administrator's Guide* for information on how to define attributes.

ObjectIdentifier

Instances of ObjectIdentifier uniquely represent objects in a 5800 system store. The 5800 system creates these instances when objects are stored and are returned to the client as part of the store result.ObjectIdentifier instances can be stored outside of the 5800 system and used

later for retrieving objects. External storage can be accomplished using an identifier's string representation by invoking the `toString` method. An instance of `ObjectIdentifier` can be reconstituted using the constructor that takes String as an argument.

`QueryResultSet`

Instances of `QueryResultSet` provide access to the objects and metadata matching a `query`. The query results can be stepped through using the `next` method. The individual results are identifiers representing objects that match the query.

If `selectKeys` was specified in the original `query`, these metadata fields can be accessed using the typed `getter` methods with each field's name.

`SystemRecord`

Instances of `SystemRecord` represent the system metadata for an object, including OID, object size, SHA1 hash, and creation time. They are returned by `storeObject` and `storeMetadata`.

`NameValueRecord`

Instances of `NameValueRecord` represent metadata used by the 5800 system to store and index user-extensible lists of name-value pairs. For convenience, instances of `NameValueRecord` also contain references to the `SystemRecord` instances of the objects they represent.

`NameValueObjectArchive` Application Access

Most applications make use of the `NameValueObjectArchive` class. This class ensures that a default metadata entry is created for every data object stored, even if no metadata is explicitly provided at store time.

The `NameValueObjectArchive` object functions as a proxy for the 5800 system server. All access is enabled by invoking methods on this object.

The following key methods and classes are used with the `NameValueObjectArchive` class:

- "`NameValueObjectArchive`" on page 31
- "`delete`" on page 31
- "`storeObject`" on page 31
- "`storeMetadata`" on page 32
- "`checkIndexed`" on page 32
- "`retrieveObject`" on page 33
- "`retrieveMetadata`" on page 33
- "`getSchema`" on page 33
- "`query`" on page 34
- "`query` (with `selectKeys`)" on page 34

NameValueObjectArchive

Initializes a new NameValueObjectArchive with the address or host name of a 5800 system server, using the provided port.

Synopsis

```
public NameValueObjectArchive(java.lang.String address)
     throws ArchiveException, java.io.IOException
public NameValueObjectArchive(String address, int port)
     throws ArchiveException, IOException
```

Description

The NameValueObjectArchive is instantiated by supplying the address of the 5800 system cluster in the constructor. The resulting data object can then be used to interact with that cluster.

delete

Deletes the metadata record.

Synopsis

```
public void
    delete(ObjectIdentifier identifier)
    throws ArchiveException, java.io.IOException
```

Description

Takes a NameValueRecord OID.

Deletes the metadata record. If it is the last metadata record referencing the underlying object data, the underlying object data will also be deleted.

storeObject

Uploads a new data object with an associated name-value metadata record.

Synopsis

```
public SystemRecord storeObject(java.nio.channels.ReadableByteChannel dataChannel)

public SystemRecord
    storeObject(ReadableByteChannel dataChannel,NameValueRecord record)
    throws ArchiveException,IOException
```

Description

Takes a `ReadableByteChannel` (and an optional `NameValueRecord`) and returns a `SystemRecord` instance containing the system metadata for the new object.

`storeMetadata`

Creates a new metadata record in the name-value object archive linked to the data object identified by the OID.

Synopsis

```
public SystemRecord storeMetadata(ObjectIdentifier linkOID,
    NameValueRecord record);
    throws ArchiveException, java.io.IOException
```

Description

Takes a `NameValueRecord` and OID and returns a `SystemRecord` instance containing the system metadata for the new metadata record.

`checkIndexed`

Checks if the metadata for an object is present in the query engine, and inserts the metadata if it is not present.

Synopsis

```
public int checkIndexed(ObjectIdentifier identifier)
        throws ArchiveException, IOException
```

Description

`checkIndexed` is intended as way to resolve a *store index exception* under program control (see "The 5800 System Query Integrity Model" on page 21 for more information).

Once a store index exception occurs (as indicated by a `SystemRecord.isIndexed` value of false after a store operation) then `archive.checkIndexed(oid)` can be called repeatedly until it returns any non-zero value. This will ensure that the metadata for the object has been inserted into the query engine; the object should then start to show up in matching queries.

`checkIndexed` returns an `int` value that indicates if the metadata for this object has been inserted into the query engine. The value is `-1` if the metadata was already inserted before this operation was called, `0` if the metadata has still not been inserted, or `1` if the metadata was just now inserted.

`retrieveObject`

Writes all of the data for the specified object into the provided channel, returning the amount of data actually retrieved.

Synopsis

```
public long retrieveObject(ObjectIdentifier oid,
    WritableByteChannel dataChannel)
    throws ArchiveException, java.io.IOException
public long retrieveObject(ObjectIdentifier oid,
    java.nio.channels.WritableByteChannel
    dataChannel,long firstByte,
    long lastByte)
    throws ArchiveException, java.io.IOException
```

Description

Takes an OID and downloads the data object into a supplied `WritableByteChannel`.

`retrieveMetadata`

Returns a `NameValueRecord` instance containing the system and name-value metadata for the metadata record identified by the OID.

Synopsis

```
public NameValueRecord
    retrieveMetadata(ObjectIdentifier oid)
    throws ArchiveException, java.io.IOException
```

Description

Returns a `NameValueRecord` instance containing the system and name-value metadata for the metadata record identified by the OID.

`getSchema`

Returns the runtime configuration of the name-value object archive as a `NameValueSchema` instance.

Synopsis

```
public NameValueSchema getSchema()
     throws ArchiveException, java.io.IOException
```

Description

Returns the runtime configuration of the name-value object archive as a `NameValueSchema` instance.

`query`

Returns a `ResultSet` of `SystemRecord` instances containing `MetadataRecord` OIDs.

Synopsis

```
public QueryResultSet
    query(java.lang.String query,int resultsPerFetch)
    throws ArchiveException, java.io.IOException
```

Description

Takes a `where` clause and returns a `QueryResultSet` of `SystemRecord` instances containing `MetadataRecord` OIDs.

The *query* parameter is a `where` clause in the 5800 system query syntax, which is a subset of SQL.

Returns a `QueryResultSet`. The results are stepped through by calling the `next` method and using the typed `get`*XXX* accessor methods.

Note – For more information on the 5800 system query language, refer to Chapter 4, "Sun StorageTek 5800 System Query Language."

`query` (with `selectKeys`)

Returns a `ResultSet` of `NameValueRecord` instances containing the selected values.

Synopsis

```
public QueryResultSet
    query(java.lang.String query, java.lang.String[] selectKeys,int maxResults)
    throws ArchiveException, java.io.IOException
```

Description

Takes a `where` clause and a `select` clause and returns a `QueryResultSet` of `NameValueRecord` instances containing the selected values.

`selectKeys` identifies the values to be returned, functioning as an SQL `select` clause.

The *query* parameter is a `where` clause in the 5800 system query syntax, which is a subset of SQL.

Returns a `QueryResultSet`. The results are stepped through by calling the `next` method and using the `getObjectIdentifier` accessor.

Note – For more information on the 5800 system query language, refer to Chapter 4, "Sun StorageTek 5800 System Query Language."

`query` (with `PreparedStatement`)

Returns the OIDs of metadata records matching the query as a `QueryResultSet` instance.

Synopsis

```
public QueryResultSet query(PreparedStatement query,
     int resultsPerFetch)
```

Description

Takes a `PreparedStatement` and returns a `QueryResultSet` of `SystemRecord` instances containing `MetadataRecord` OIDs.

The `PreparedStatement` parameter enables queries with dynamic parameters to pass typed data items to the query.

Returns a `QueryResultSet`. The results are stepped through by calling the `next` method and using the typed `get`*XXX* accessor methods.

Note – For more information on the 5800 system query language, refer to Chapter 4, "Sun StorageTek 5800 System Query Language."

`query` (with `PreparedStatement` and `selectKeys`)

Returns specified fields from metadata records matching the query as a `QueryResultSet` instance.

Synopsis

```
public QueryResultSet
    query(PreparedStatement query,
    java.lang.String[] selectKeys,
    int resultsPerFetch)
```

Description

Takes a `where` clause and a `select` clause and returns a `QueryResultSet` of `NameValueRecord` instances containing the selected values.

`selectKeys` identifies the values to be returned, functioning as an SQL `select` clause.

The *PreparedStatement* parameter enables queries with dynamic parameters to pass typed data items to the query.

Returns a `QueryResultSet`. The results are stepped through by calling the `next` method and using the `getObjectIdentifier` accessor.

Note – For more information on the 5800 system query language, refer to Chapter 4, "Sun StorageTek 5800 System Query Language."

`PreparedStatement`

Extends `com.sun.honeycomb.common.Encoding`

Synopsis

```
public PreparedStatement(java.lang.String sql);
```

Description

Used to implement queries with Dynamic Parameters, which is the preferred way to pass typed data items to a StorageTek 5800 query.

The number of `bindParameter` calls should match the number of question marks (`?`) in the query string in the prepared statement. Parameters are specified positionally. For example, a `bindParameter` call with `index = 1` supplies a value for the first `?` in the supplied query string. Once a value has been supplied for each of the dynamic parameters, then the `PreparedStatement` may be passed to the `NameValueObjectArchive.query` method to be executed, for example:

```
NameValueObjectArchive archive = new NameValueObjectArchive(hostname);
Date date_value= new java.sql.Date();
PreparedStatement stmt = new PreparedStatement("date_field<?");
```

```
stmt.bindParameter(date_value,1);
QueryResultSet qrs = archive.query(stmt);
```

QueryResultSet

The QueryResultSet class is used to page through OIDs and associated metadata returned by NameValueObjectArchive.query. See the javadoc for the getXXX methods for getting typed metadata.

next

Sets the QueryResultSet to point at the next record.

Synopsis

```
boolean next()
```

Description

Sets the QueryResultSet to point at the next record. Returns true if there is a next record, false if not.

getObjectIdentifier

Gets the ObjectIdentifier of the current metadata record.

Synopsis

```
ObjectIdentifier getObjectIdentifier()
```

Description

Gets the ObjectIdentifier of the current metadata record.

isQueryComplete

Returns whether the set of results constitutes a complete set

Synopsis

```
boolean isQueryComplete()
```

Description

Returns whether the set of results constitutes a complete set. See "The 5800 System Query Integrity Model" on page 21.

getQueryIntegrityTime

Returns the most recent time at which all store index exceptions are known to have been resolved.

Synopsis

```
long getQueryIntegrityTime()
```

Description

The query integrity time is a time such that all store index exceptions from before that time have been resolved. There is an ideal query integrity time, which is the time of the oldest still-unresolved store index exception: an ideal implementation when asked for the query integrity time would always report this ideal value. In actual implementation, the reported query integrity time might be hours or even days earlier than the ideal query integrity time, depending on how far the ongoing system healing has progressed.

QueryIntegrityTime

Get detailed status on which store index exceptions might still be unresolved

Synopsis

```
QueryResultSet.isQueryComplete(), QueryResultSet.getQueryIntegrityTime();
```

◆ ◆ ◆ CHAPTER 3

Sun StorageTek 5800 System C Client API

This chapter provides detailed information on the 5800 system C client API.

The following topics are discussed:

Overview of the 5800 System C Client API

This section provides an overview of the 5800 system C client API. The following topics are discussed:

Architecture

The 5800 system C API client supports two different access patterns: a synchronous "EZ" access very similar to the current Java implementation, and a more flexible, nonblocking access based on the POSIX model.

Note – For this release, the nonblocking C API client is not implemented.

Interfaces

The C client library interacts with the 5800 system server entirely through an HTTP protocol.

Retrying Operations

Calls to the C API should be wrapped with retry logic so that their applications are resilient to transient failures that may be experienced when a node or switch fails while servicing an operation.

Multithreaded Access

Both the synchronous and the nonsynchronous C APIs are fully thread-safe and can be used simultaneously in multiple threads from the same process. Each thread must call "`hc_session_create_ez`" on page 53 to create its own session. Sessions must not be shared between threads.

Caution – Name resolution must be done in a single thread with the subsequent IP address passed to `hc_session_create_ez`, otherwise core dumps will occur if multiple name resolution threads call `getaddrinfo` at the same time.

Performance and Scalability

The 5800 system C client library provides high performance and is highly scalable.

The synchronous C API performs its own calls to `select()` internally.

For the nonblocking C API (not yet implemented), access is provided to the underlying `fd_set` so that all pending I/O operations can be serviced by a single thread on the basis of status returned by the POSIX `select()` function, possibly after merging the 5800 system `fd_set` with some external, application-specific `fd_set`.

Memory Usage

The 5800 system C client library generally follows the model of populating externally allocated data structures such as handles, buffers, and result arrays.

Some internal data structures are generated during XML document construction. These data structures are allocated and freed using the function pointers supplied to `hc_init` when initializing the library (see "Initializing a Global Session" on page 41).

Other data structures are allocated and returned to the user; these have corresponding functions to free them. For example, `hc_session_create_ez` and `hc_session_free`.

Updating Schema Definitions

The C client library does not automatically refresh its in-memory schema definitions. If the schema is changed through the command-line interface (CLI), a new session must be created with a new call to "`hc_session_create_ez`" on page 53 to access the schema changes.

Session Management

A global session must be explicitly initialized with a call to `hc_init` and released with a call to `hc_cleanup`. Memory allocators and deallocators are supplied in the initialization function to control how memory allocation occurs. You will normally supply the standard `malloc`, `free`, and `realloc` functions for this functionality.

Heap Memory Allocator

The heap memory allocator is defined as follows:

```
typedef void* (*allocator_t) (size_t size);
```

Heap Memory Deallocator

The heap memory deallocator is defined as follows:

```
typedef void (*deallocator_t) (void *p);
```

Heap Memory Reallocator

The heap memory reallocator is defined as follows:

```
typedef void (*reallocator_t) (void *p,size_t size);
```

Initializing a Global Session

The following function initializes a global session:

```
hcerr_t hc_init(allocator_t,
      deallocator_t,
      reallocator_t);
```

This function must be called once per process to initialize the memory functions used in the 5800 system C API. It also initializes global session properties.

A global session is initialized once per process, regardless of how many threads in that process are using the C API.

Note – `hc_init` should be called once per process before any thread calls "hc_session_create_ez" on page 53. If "hc_session_create_ez" on page 53 is called before `hc_init`, an implicit call is made to `hc_init` from that thread. But that call to `hc_init` is not interlocked with other threads, and it uses the C API shared library's version of `malloc` and `free`, which might be different than the application's version of `malloc` and `free`. It is strongly recommended that all applications call `hc_init` once per process with their own allocator and deallocator.

Note – For more information on "hc_session_create_ez" on page 53 , see "hc_session_create_ez" on page 53.

Terminating a Global Session

The following function terminates a global session:

```
void hc_cleanup();
```

System Record

All 5800 system `store` operations return a system record, which encapsulates information about the stored object. In particular, the system record contains the OID, which can be used to retrieve the stored object data or metadata.

```
typedef struct hc_system_record_ {
    char is_indexed;
    hc_oid oid;
    hc_digest_algo digest_algo;
    hc_digest data_digest;
    hc_long_t size;
    hc_long_t creation_time;
    hc_long_t deleted_time;
    char shredMode;
} hc_system_record_t;
```

About the fields:

- **oid** — The objectid for this object, equivalent to the `system.object_id` field.
- **digest_algo** — Always set to "sha1" for this release. Equivalent to the `system.object_hash_alg` field.
- **data_digest** — An array of bytes that represent the content digest of this object's data. Equivalent to the `system.object_hash` field.
- **size** — The size of the data in this object, in bytes. Equivalent to the `system.object_size` field.
- **creation_time** — The object creation time, expressed as number of milliseconds since the epoch. Equivalent to the `system.object_ctime` field.
- **deleted_time** — The deletion time of this record, if any, as the number of milliseconds since the epoch.
- **shredMode** — Not used in this release.
- **is_indexed** — indicates, after a `store_data` or `store_metadata` operation, whether the metadata for the object was successfully inserted into the query engine, and the object is hence available for query. `0` if false, `1` if true.

Failure and Recovery

Every function in the 5800 system C client library returns a result code of type `hcerr_t`. Any value other than `HCERR_OK` indicates a nonrecoverable error. See the `hc.h` file for specific error codes.

C Client Application Deployment

C applications using the 5800 system C API use both the 5800 system libraries and the `curl` libraries. These libraries are different for each supported platform (Windows, Linux, Solaris (x86), Solaris (SPARC)) and are located in the `c/<OS>/lib` directory in the SDK.

Note – The environment variable `http_proxy` should not be set for processes using the C API, since the HTTP client library (`curl`) makes use of it.

Nonblocking C API

The nonblocking C API is not implemented for this release of the 5800 system. If you are interested in working with the nonblocking C API, contact your 5800 system Sales Representative.

Synchronous C API

A multiplatform synchronous C API in which operations are accomplished in a few simple function calls is provided for the 5800 system. The API calls include operations for storing, retrieving, deleting, and querying of data and metadata records. Multiple threads are supported, and operations block until they complete.

You must call `hc_init` (once per process) and "`hc_session_create_ez`" on page 53 (once per thread) prior to making any other API calls.

All functions in the 5800 system C API return an `hc_err`. Any value other than `HCERR_OK` indicates failure.

This section discusses the following topics for the 5800 system synchronous C API.

Changes for the 1.1 Release

This release of the synchronous C API contains the following changes:

- Handling is added for storing, retrieving and querying the following new metadata types:
  - char — for Latin 1 character set
  - unicode
  - binary
  - date
  - time
  - timestamp

  Query and queryplus are merged
- Prepared statements (`pstmts`) are introduced to handle values of queries that cannot be placed inline, and a new query is introduced to handle them.
- The following new functions have been added to the API:
  - "`hc_check_indexed_ez`" on page 89
  - "`hc_decode_hcerr`" on page 112
  - "`hc_decode_hc_type`" on page 113
  - "`hc_schema_get_length`" on page 61
  - "`hc_nvr_add_binary`" on page 71
  - "`hc_nvr_add_date`" on page 72
  - "`hc_nvr_add_time`" on page 73
  - "`hc_nvr_add_timestamp`" on page 74
  - "`hc_nvr_get_binary`" on page 81
  - "`hc_nvr_get_date`" on page 82

- "hc_nvr_get_time" on page 82
- "hc_nvr_get_timestamp" on page 83
- "hc_pstmt_create" on page 100
- "hc_pstmt_free" on page 101
- "hc_pstmt_set_string" on page 101
- "hc_pstmt_set_char" on page 102
- "hc_pstmt_set_double" on page 103
- "hc_pstmt_set_long" on page 104
- "hc_pstmt_set_date" on page 105
- "hc_pstmt_set_time" on page 106
- "hc_pstmt_set_timestamp" on page 107
- "hc_pstmt_set_binary" on page 108
- "hc_pstmt_query_ez" on page 109
- "hc_qrs_is_query_complete" on page 97
- "hc_qrs_get_query_integrity_time" on page 98

The following functions have changed in the API:

- "hc_query_ez" on page 94
- "hc_qrs_next_ez" on page 96

The following functions and types have been removed from the API:

- hc_query_plus_result_set_t
- hc_query_plus_ez
- hc_qprs_next_ez
- hc_qprs_free

Limitations

This release of the synchronous C API is subject to the following limitations:

- Changes to the metadata schema at the server are only detected at the client at the next call to "hc_session_create_ez" on page 53.
- The values returned by "hc_session_get_platform_result" on page 58 will not be updated properly after calls to the functions "hc_retrieve_ez" on page 91 and "hc_delete_ez" on page 111.
- When using the C API, the maximum metadata size of a data item stored using either "hc_store_both_ez" on page 87 or "hc_store_metadata_ez" on page 88 is limited to approximately 76300 bytes. The exact maximum metadata size depends on many factors and should not be relied on. This limitation does not apply to metadata stored using the Java API.

Synchronous C Data Types

The following data types are defined for the C API:

hc_string_t

Type for holding Unicode (UTF-8) and Latin-1 null-terminated strings.

Synopsis

```
typedef char *hc_string_t;
```

Description

This type is used interchangeably for holding Unicode (UTF-8) and Latin-1 null-terminated metadata strings. The context determines whether the contents are UTF-8 or Latin-1.

hc_long_t

Type for holding integer values.

Synopsis

```
typedef int64_t hc_long_t;
```

Description

Type for holding integer values.

hc_double_t

Type for holding floating-point values.

Synopsis

```
typedef double hc_double_t;
```

Description

Type for holding floating-point values.

hc_type_t

5800 system name-value metadata type specifier.

Synopsis

```
typedef enum hc_types_{
    HC_UNKNOWN_TYPE = -1,
    HC_BOGUS_TYPE = 0,
    HC_STRING_TYPE = 1,
    HC_LONG_TYPE = 2,
    HC_DOUBLE_TYPE = 3,
    HC_BYTE_TYPE = 4,
    HC_CHAR_TYPE = 5,
    HC_BINARY_TYPE = 6,
    HC_DATE_TYPE = 7,
    HC_TIME_TYPE = 8,
    HC_TIMESTAMP_TYPE = 9,
    HC_OBJECTID_TYPE = 10,
} hc_type_t;
```

Description

Specifies one of the 5800 system metadata types that can go in the archive.

hc_value_t

5800 system name-value metadata data value.

Synopsis

```
typedef struct hc_value_ {
    hc_type_t hcv_type;
    union {
        hc_string_t hcv_string;
        hc_long_t hcv_long;
        hc_double_t hcv_double;
```

```
        hc_bytearray_t hcv_bytearray;
        struct tm hcv_tm;
        struct timespec hcv_timespec;
    } hcv;
} hc_value_t;
```

Description

This tagged union type can be used to hold a reference to any of the 5800 system data types.

hc_schema_t

5800 system name-value metadata schema.

Synopsis

```
typedef void hc_schema_t;
```

Description

An opaque structure that holds the names and data types of each element in the archive's metadata schema.

hc_nvr_t

5800 system name-value record.

Synopsis

```
typedef void hc_nvr_t;
```

Description

An opaque structure to represent one metadata record. There is a count of metadata tuples, and parallel sets of names and of typed values for the tuples in this metadata record.

hc_session_t

Structure describing the connection from one thread to one 5800 system server.

Synopsis

```
typedef void hc_session_t;
```

Description

An opaque structure to represent the session from one thread to one 5800 system server. It contains the schema used to interpret metadata store and retrieve operations to this 5800 system server.

hc_pstmt_t

Structure for holding a prepared statement.

Synopsis

```
typedef void hc_pstmt_t;
```

Description

An opaque structure representing a query, including the query text and bound fields.

hc_query_result_set_t

Structure used to hold the results of a query.

Synopsis

```
typedef void hc_query_result_set_t;
```

Description

This opaque structure is used to hold the results of a query. For more information on the functions that use this structure, see "Querying Metadata" on page 94.

read_from_data_source

Data source template used to upload object data to the cluster.

Synopsis

```
typedef long (*read_from_data_source)
(void *cookie, char *buf, long buf_size);
```

Description

Function pointers of `read_from_data_source` type are used to upload object data. The function pointer and opaque `cookie` reference are supplied as arguments to "`hc_store_both_ez`" on page 87 and other functions that store object data. The data source reader function will be called repeatedly, with the supplied `cookie` as an argument, to gather the object data to upload into storage.

A `read_from_data_source` function should read up to `buf_size` bytes from the data source indicated by `cookie` into the buffer at location `buff` and return the actual number of bytes read as the return value from the function.

There are two special return codes:

- A return value of `0` indicates the end-of-file condition. The data should be committed to the data store.
- A return value of `-1` indicates a request to cancel the store. The store operation should be aborted with an error code of `HCERR_ABORTED_BY_CALLBACK`.

Parameters

`cookie`

An opaque data structure to identify this data `cookie`. The `cookie` is likely to be an open file descriptor.

`buf`

Where to store the data.

`buf_size`

The number of available bytes of space in `buf`.

See Also

"`hc_store_both_ez`" on page 87

write_to_data_destination

Data destination template used to download object data to the cluster.

Synopsis

```
typedef long (*write_to_data_destination)
(void *cookie, char *buff, long buff_len);
```

Description

Function pointers of `write_to_data_destination` type are used to download object data to a network or other destination from the 5800 system server using "hc_retrieve_ez" on page 91. The function pointer and opaque `cookie` reference are supplied as arguments to "hc_retrieve_ez" on page 91, and the function will be called with the supplied `cookie` argument to deliver the downloaded data to a local data storage function.

A `write_to_data_destination` function should write exactly `buff_len` bytes to the data destination indicated by `cookie`, reading the bytes from the buffer at location `buff`. It should return a `long` value indicating the number of bytes actually processed. A return code that differs from `buff_len` indicates that the transfer should be terminated.

Parameters

`cookie`	An opaque data structure to identify this data `cookie`. The `cookie` is likely to be an open file descriptor.
`buff`	Where to copy the data from.
`buff_len`	The number of bytes of space in `buff`.

See Also

"hc_retrieve_ez" on page 91

hcerr_t

5800 system C client API error codes.

To decode `hcerr_t` values into strings, see "hc_decode_hcerr" on page 112

Synopsis

```
typedef enum hcerr {
    HCERR_OK = 0,
    HCERR_NOT_INITED,
    HCERR_ALREADY_INITED,
    HCERR_INIT_FAILED,
    HCERR_OOM,
    HCERR_NOT_YET_IMPLEMENTED,
    HCERR_SESSION_CREATE_FAILED,
    HCERR_ADD_HEADER_FAILED,HCERR_IO_ERR,
    HCERR_FAILOVER_OCCURRED,
    HCERR_CAN_CALL_AGAIN,
    HCERR_GET_RESPONSE_CODE_FAILED,
    HCERR_CONNECTION_FAILED,
```

```
    HCERR_BAD_REQUEST,
    HCERR_NO_SUCH_OBJECT,
    HCERR_INTERNAL_SERVER_ERROR,
    HCERR_FAILED_GETTING_FDSET,
    HCERR_FAILED_CHECKING_FDSET,
    HCERR_MISSING_SELECT_CLAUSE,
    HCERR_URL_TOO_LONG,
    HCERR_COULD_NOT_OPEN_FILE,
    HCERR_FAILED_TO_WRITE_TO_FILE,
    HCERR_NULL_SESSION,
    HCERR_INVALID_SESSION,
    HCERR_INVALID_OID,
    HCERR_NULL_HANDLE,
    HCERR_INVALID_HANDLE,
    HCERR_INVALID_SCHEMA,
    HCERR_INVALID_RESULT_SET,
    HCERR_INVALID_NVR,
    HCERR_WRONG_HANDLE_FOR_OPERATION,
    HCERR_HANDLE_IN_WRONG_STATE_FOR_OPERATION,
    HCERR_READ_PAST_LAST_RESULT,
    HCERR_XML_PARSE_ERROR,
    HCERR_XML_MALFORMED_XML,
    HCERR_XML_EXPECTED_LT,
    HCERR_XML_INVALID_ELEMENT_TAG,
    HCERR_XML_MALFORMED_START_ELEMENT,
    HCERR_XML_MALFORMED_END_ELEMENT,
    HCERR_XML_BAD_ATTRIBUTE_NAME,
    HCERR_XML_BUFFER_OVERFLOW,
    HCERR_BUFFER_OVERFLOW,
    HCERR_NO_SUCH_TYPE,
    HCERR_ILLEGAL_VALUE_FOR_METADATA,
    HCERR_NO_SUCH_ATTRIBUTE,
    HCERR_NO_MORE_ATTRIBUTES,
    HCERR_EOF,HCERR_FAILED_GETTING_SILO_DATA,
    HCERR_PLATFORM_NOT_INITED,
    HCERR_PLATFORM_ALREADY_INITED,
    HCERR_PLATFORM_INIT_FAILED,
    HCERR_PLATFORM_HEADER_TOO_LONG,
    HCERR_PLATFORM_TOO_LATE_FOR_HEADERS,
    HCERR_PLATFORM_NOT_ALLOWED_FOR_GET,
    HCERR_FAILED_TO_GET_SYSTEM_RECORD,
    HCERR_PARTIAL_FILE,
    HCERR_ABORTED_BY_CALLBACK,
    HCERR_PLATFORM_GENERAL_ERROR,
    HCERR_ILLEGAL_ARGUMENT
} hcerr_t;
```

Description

This structure defines the 5800 system C client API error codes.

Synchronous C API Functions

The 5800 system synchronous C API functions are defined to perform the following tasks:

Managing 5800 System Sessions

The following functions are used to manage 5800 system sessions:

hc_session_create_ez

Creates a session.

Synopsis

```
hcerr_t hc_session_create_ez(char *host,
      int port,
      hc_session_t **sessionp);
```

Description

This function initializes the 5800 system API and must be called before calling any of the other functions in this API. It downloads a copy of the schema for a particular host or port. The schema is used to validate the name-value-type tuples that are added to metadata records.

Both the synchronous and the nonsynchronous C APIs are fully thread-safe and can be used simultaneously in multiple threads from the same process. Each thread must call "hc_session_create_ez" on page 53 to create its own session. Sessions must not be shared between threads.

Note – `hc_init` should be called once per process before any thread calls `hc_session_create_ez`. If `hc_session_create_ez` is called before `hc_init`, an implicit call is made to `hc_init` from that thread. But that call to `hc_init` is not interlocked with other threads, and it uses the C API shared library's version of `malloc` and `free`, which might be different than the application's version of `malloc` and `free`. It is strongly recommended that all applications call `hc_init` once per process with their own allocator and deallocator.

For more information on `hc_init`, see "Initializing a Global Session" on page 41

Parameters

`host`

IN: The name or IP address of a 5800 system server.

`port`

IN: The port number of the 5800 system server (normally 8080).

`sessionp`

OUT: Updated to point to a session object.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_ILLEGAL_ARGUMENT
```

hc_session_free

Releases the session object.

Synopsis

```
hcerr_t hc_session_free (hc_session_t *session);
```

Description

This function releases the session object and recovers handles and memory for one connection.

Parameters

`session`

IN: The session object to free.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_NULL_SESSION
HCERR_INVALID_SESSION
```

hc_session_get_status

Gets the session status.

Synopsis

```
hcerr_t hc_session_get_status(hc_session_t *session,
      int32_t *response_codep,char **errstrp);
```

Description

This function returns the HTTP response code and the error message string associated with the last request on this session.

Parameters

`session`

IN: The session object.

`response_codep`

OUT: Updated to be the HTTP response code.

`errstr`

IN: Updated to be the error returned in the response body if the response code is not 200 (OK). `errstr` should not be written to by the application (that is, it is read only), and will persist until the next request to the 5800 system server or until "`hc_session_free`" on page 55 is called, whichever comes first.

Return Codes

```
 HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_NULL_SESSION
HCERR_INVALID_SESSION
```

hc_session_get_schema

Gets schema object associated with this session.

Synopsis

```
hcerr_t hc_session_get_schema (hc_session_t *session,
     hc_schema_t **schemap);
```

Description

This function returns the current schema object associated with this session.

Parameters

`session`

IN: The session object.

`schemap`

OUT: Updated to be the schema associated with the current session. `schemap` should not be modified by the application and will persist until the next call to "`hc_session_free`" on page 55, at which time it will be implicitly released.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_NULL_SESSION
HCERR_INVALID_SESSION
```

hc_session_get_host

Returns the host name and port number associated with the session.

Synopsis

```
hc_session_get_host(hc_session_t *session,
     char **hostp,int *portp);
```

Description

This function returns the host name and port number associated with the session.

Parameters

`session`

IN: The session object.

`hostp`

OUT: Updated to point to host name (read-only string). `hostp` should not be modified by the application and will persist until the next call to "`hc_session_free`" on page 55.

`portp`

OUT: Updated to be the port number.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_NULL_SESSION
HCERR_INVALID_SESSION
```

hc_session_get_platform_result

Returns low-level error codes associated with the current session.

Synopsis

```
hcerr_t hc_session_get_platform_result(hc_session_t *session,
     int32_t *connect_errnop,
     int32_t *platform_resultp);
```

Description

This function returns low-level error codes associated with the current session.

Note – The values returned by `hc_session_get_platform_result` will not be updated properly after calls to the functions `hc_retrieve_ez` and `hc_delete_ez`.

Parameters

`session`

IN: The session object.

`connect_errnop`

OUT: Updated to be the operating system's `errno` value associated with the last connect operation on the current session.

`platform_resultp`

OUT: Updated to be the error code reported by the underlying HTTP library (for example, the `Curl` library).

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_NULL_SESSION
HCERR_INVALID_SESSION
```

hc_session_get_archive

Returns the current archive object associated with this session.

Synopsis

```
hcerr_t hc_session_get_archive(hc_session_t *session,
     hc_archive_t **archivep);
```

Description

This function returns the current archive object associated with this session.

Note – The archive object is not needed for the synchronous C API, but is made available for interfacing with the lower-level (nondocumented) API.

Parameters

`session`

IN: The session object.

`archivep`

OUT: Updated to be the archive object associated with the current session.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_NULL_SESSION
HCERR_INVALID_SESSION
```

Managing a Schema

When a session commences, the C client API downloads information about the metadata schema that is in use on the 5800 system server.

The following functions are used to manage a schema:

- "hc_schema_get_type" on page 60
- "hc_schema_get_length" on page 61
- "hc_schema_get_count" on page 61

- "hc_schema_get_type_at_index" on page 62

hc_schema_get_type

Looks up type in schema.

Synopsis

```
hcerr_t hc_schema_get_type(hc_schema_t *schema,
     char *name, hc_type_t *typep);
```

Description

This function looks up the type associated with a given name in the current metadata schema, or returns an error if the name is not known.

Parameters

`schema`

IN: The schema to interrogate.

`name`

IN: The attribute name to look up in the schema.

`typep`

OUT: Updated to be the type associated with that name in the schema.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_INVALID_SCHEMA
HCERR_ILLEGAL_ARGUMENT
```

hc_schema_get_length

Looks up length of `char` and `string` attribute fields.

Synopsis

```
hcerr_t hc_schema_get_length(hc_schema_t *schema,
     char *name, int *length);
```

Description

This function looks up the length of a `char` or `string` field associated with a given attribute name in the current metadata schema, or returns an error if the name is not known.

Parameters

`schema`

IN: The schema to interrogate.

`name`

IN: The attribute name to look up in the schema.

`length`

OUT: Updated to be the length of the field associated with that name in the schema.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_INVALID_SCHEMA
HCERR_ILLEGAL_ARGUMENT
```

hc_schema_get_count

Returns the number of name-value pairs in the metadata schema.

Synopsis

```
hcerr_t hc_schema_get_count(hc_schema_t *hsp,
     hc_long_t *countp);
```

Description

This function returns the number of name-value pairs in the metadata schema.

Parameters

`hsp`

IN: The schema to interrogate.

`countp`

OUT: Updated with the number of name-value pairs in the schema.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_INVALID_SCHEMA
```

See Also

"hc_schema_get_type_at_index" on page 62

hc_schema_get_type_at_index

Iterates through the name-value pairs in a schema.

Synopsis

```
hcerr_t hc_schema_get_type_at_index (hc_schema_t *hsp,
     hc_long_t index,char **namep,
     hc_type_t *typep);
```

Description

This function provides a simple way to iterate through the name-value pairs in a schema.

Parameters

`hsp`

IN: The schema to query.

`index`

IN: Should range from `0` up to the `count-1` returned in "hc_schema_get_count" on page 61.

`namep`

OUT: Updated to point to a string that is an attribute name of one attribute in the schema.

`typep`

OUT: Updated to be the type associated with that name in the schema. If the server schema references a type that the client library does not support, then the type is returned as `HC_UNKNOWN_TYPE`.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_INVALID_SCHEMA
HCERR_ILLEGAL_ARGUMENT
```

See Also

"hc_schema_get_count" on page 61

Manipulating Name-Value Records

5800 system synchronous C API functions are defined to perform the following name-value record manipulation tasks:

Using the API for Storing Name-Value Records

A common way of storing metadata in the synchronous C API for the 5800 system is to use the name-value record API.

▼ To Use the API for Storing Name-Value Records

1. **Call `hc_init` once per process.**
2. **Call "`hc_session_create_ez`" on page 53 to initialize the session and download the schema.**
3. **Create the metadata record with "`hc_nvr_create`" on page 65.**
4. **Fill the new metadata piece by piece with `hc_nvr_add_metadata_*` functions (see "Building Name-Value Records" on page 66) for each 5800 system type.**
5. **Call either "`hc_store_metadata_ez`" on page 88 or "`hc_store_both_ez`" on page 87 to store the new metadata record.**
6. **When you are done, free the metadata record by calling "`hc_nvr_free`" on page 66.**
7. **When the session is finished, call "`hc_session_free`" on page 55 to free the session data structures.**
8. **When all threads are completed, call `hc_cleanup` to release the global session.**

Using Returned Name-Value Records

Name-value records are also returned as the result of queries that return metadata information, such as "`hc_retrieve_metadata_ez`" on page 92.

▼ To Use Returned Name-Value Records

1. **Run the query to create an "`hc_nvr_t`" on page 48 record or a table of "`hc_nvr_t`" on page 48 structures.**

   Use either name-based access (for example, `hc_nvr_get_*`) or index-based access (for example, "`hc_nvr_get_count`" on page 76 and "`hc_nvr_get_value_at_index`" on page 77).

2. **To free the "`hc_nvr_t`" on page 48 structure, call "`hc_nvr_free`" on page 66.**

Note – Structures created by `hc_nvr_create` can also be freed by calling `hc_nvr_free`.

Creating and Freeing Name-Value Records

The following functions are defined to create and free name-value records:

hc_nvr_create

Creates a name-value record.

Synopsis

```
hcerr_t hc_nvr_create(hc_session_t *session,
     hc_long_t nslots,
     hc_nvr_t **nvrp);
```

Description

This function creates a name-value record with a designated initial size that is associated with a particular session. Metadata that is added to this name-value record must match the schema associated with the session.

Parameters

`session`

IN: The session with which this name-value record is associated.

`nslots`

IN: The number of slots for name-value-type tuples.

`nvrp`

OUT: Updated with a pointer to a new name-value record of the designated size.

Return Codes

```
HCERR_OK
HCERR_ILLEGAL_ARGUMENT
HCERR_OOM
```

See Also

"hc_nvr_free" on page 66

hc_nvr_free

Frees a name-value record.

Synopsis

```
hcerr_t hc_nvr_free(hc_nvr_t *nvr);
```

Description

This function frees a name-value record that was created by "hc_nvr_create" on page 65.

Parameter

`nvr`

IN: Points to the name-value-record to be freed.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_INVALID_NVR
```

See Also

"hc_nvr_create" on page 65

Building Name-Value Records

The following functions are defined to build name-value records:

- "hc_nvr_add_value" on page 67
- "hc_nvr_add_long" on page 68
- "hc_nvr_add_double" on page 69
- "hc_nvr_add_string" on page 70
- "hc_nvr_add_binary" on page 71
- "hc_nvr_add_date" on page 72
- "hc_nvr_add_time" on page 73

- "hc_nvr_add_timestamp" on page 74
- "hc_nvr_add_from_string" on page 75

hc_nvr_add_value

Adds a new metadata value.

Synopsis

```
hcerr_t hc_nvr_add_value(hc_nvr_t *nvr,
     char *name, hc_value_t value);
```

Description

This function adds a new metadata name-value-type tuple to a designated name-value record. The name-value record will automatically expand as needed.

Parameters

`nvr`

Points to a name-value-record.

`name`

IN: Name for the tuple.

`value`

IN: Value for the tuple, in the type-tagged "hc_value_t" on page 47 format.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_INVALID_NVR
HCERR_ILLEGAL_ARGUMENT
HCERR_ILLEGAL_VALUE_FOR_METADATA
HCERR_NO_SUCH_ATTRIBUTE
```

hc_nvr_add_long

Adds a new metadata value of type hc_long_t.

Synopsis

```
hcerr_t hc_nvr_add_long(hc_nvr_t *nvr,
     char *name, hc_long_t value)
```

Description

This function adds a new metadata name-value-type tuple to a designated name-value record, where type is known to be hc_long_t (see "hc_type_t" on page 47). The name-value record will automatically expand as needed.

Parameters

nvr

Points to a name-value-record.

name

IN: Name for the tuple.

value

IN: The hc_long_t value.

Return Codes

HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_INVALID_NVR
HCERR_ILLEGAL_ARGUMENT
HCERR_ILLEGAL_VALUE_FOR_METADATA
HCERR_NO_SUCH_ATTRIBUTE

hc_nvr_add_double

Adds a new metadata value of type `hc_double_t`.

Synopsis

```
hcerr_t hc_nvr_add_double(hc_nvr_t *nvr,
    char *name,
    hc_double_t value);
```

Description

This function adds a new metadata name-value-type tuple to a designated name-value record, where type is known to be `hc_double_t` (see "hc_type_t" on page 47). The name-value record will automatically expand as needed.

Parameters

`nvr`

Points to a name-value-record.

`name`

IN: Name for the tuple.

`value`

IN: The `hc_double_t` value.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_INVALID_NVR
HCERR_ILLEGAL_ARGUMENT
HCERR_ILLEGAL_VALUE_FOR_METADATA
HCERR_NO_SUCH_ATTRIBUTE
```

hc_nvr_add_string

Adds a new metadata value of type Unicode UTF-8 string.

Synopsis

```
hcerr_t hc_nvr_add_string(hc_nvr_t *nvr,
    char *name,
    hc_string_t value);
```

Description

This function adds a new metadata name-value-type tuple to a designated name-value record, where type is a Unicode UTF-8 string. The name-value record automatically expands as needed. The string is copied into the structure.

Parameters

`nvr`

Points to a name-value-record.

`name`

IN: Name for the tuple.

`value`

IN: The `hc_string_t` value.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_INVALID_NVR
HCERR_ILLEGAL_ARGUMENT
HCERR_ILLEGAL_VALUE_FOR_METADATA
HCERR_NO_SUCH_ATTRIBUTE
```

hc_nvr_add_binary

Adds new metadata value of type `binary`.

Synopsis

```
hcerr_t hc_nvr_add_binary(hc_nvr_t *nvr,
     hc_string_t name,
     int size,
     unsigned char *bytes);
```

Description

This function adds a new metadata name-value-type tuple to a designated name-value record, where type is binary data. The name-value record automatically expands as needed. The name and data are copied into the structure.

Parameters

`nvr`

Points to a name-value-record.

`name`

IN: Name for the tuple.

`size`

IN: The size of the data.

`bytes`

IN: The binary data.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_INVALID_NVR
HCERR_ILLEGAL_ARGUMENT
HCERR_ILLEGAL_VALUE_FOR_METADATA
HCERR_NO_SUCH_ATTRIBUTE
```

hc_nvr_add_date

Adds new metadata value of type `date`.

Synopsis

```
#include <time.h>
hcerr_t hc_nvr_add_date(hc_nvr_t *nvr,
     hc_string_t name,struct tm *value);
```

Description

This function adds a new metadata name-value-type tuple to a designated name-value record.

The `struct tm` fields are as defined in the POSIX standard and interpreted by `mktime(3C)`. All fields are ignored except:

```
int tm_mday; /* day of the month - [1, 31] */
int tm_mon; /* months since January - [0, 11] */
int tm_year; /* years since 1900 */
```

The name-value record automatically expands as needed. The name and value are copied into the structure.

Parameters

`nvr`

Points to a name-value-record.

`name`

IN: Name for the tuple.

`value`

IN: The `struct tm` (`time.h`) value.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_INVALID_NVR
HCERR_ILLEGAL_ARGUMENT
HCERR_ILLEGAL_VALUE_FOR_METADATA
HCERR_NO_SUCH_ATTRIBUTE
```

hc_nvr_add_time

Adds new metadata value of type `time`.

Synopsis

```
#include <time.h>
hcerr_t hc_nvr_add_time(hc_nvr_t *nvr,
     hc_string_t name,
     time_t *value);
```

Description

This function adds a new metadata name-value-type tuple to a designated name-value record. The value represents seconds since midnight.

The name-value record automatically expands as needed. The name and value are copied into the structure.

Parameters

`nvr`

Points to a name-value-record.

`name`

IN: Name for the tuple.

`value`

IN: The `time_t` (`time.h`) value.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_INVALID_NVR
HCERR_ILLEGAL_ARGUMENT
HCERR_ILLEGAL_VALUE_FOR_METADATA
HCERR_NO_SUCH_ATTRIBUTE
```

hc_nvr_add_timestamp

Adds new metadata value of type `timestamp`.

Synopsis

```
#include <time.h>
hcerr_t hc_nvr_add_timestamp(hc_nvr_t *nvr,
    hc_string_t name,
    struct timespec *value);
```

Description

This function adds a new metadata name-value-type tuple to a designated name-value record, where type is `hc_timestamp_t`. The `struct timespec` is defined in the POSIX standard:

```
time_t tv_sec; /* seconds */
long tv_nsec; /* and nanoseconds */
```

where `tv_sec` is measured since the UNIX epoch (00:00:00 UTC on January 1, 1970). The maximum value of `tv_sec` is truncated by three decimal digits owing to database limitations and `tv_nsec` is truncated to milliseconds. The name-value record automatically expands as needed. The name and value are copied into the structure.

Parameters

`nvr`

Points to a name-value-record.

`name`

IN: Name for the tuple.

`value`

IN: The '`struct timespec` (`time.h`) value.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_INVALID_NVR
HCERR_ILLEGAL_ARGUMENT
HCERR_ILLEGAL_VALUE_FOR_METADATA
HCERR_NO_SUCH_ATTRIBUTE
```

hc_nvr_add_from_string

Adds a new metadata value where the value always starts out as a string.

Synopsis

```
hcerr_t hc_nvr_add_from_string(hc_nvr_t *nvr,
      char *name,
      char *value);
```

Description

This is a convenient function for adding a new metadata name-value-type tuple to a designated name-value, where the value always starts out as a string. The correct metadata type for `name` must be looked up from the schema. The name-value record will automatically expand as needed. The string is copied into the structure.

Parameters

`nvr`

Points to a name-value-record.

`name`

IN: Name for the tuple.

`value`

IN: The string value to be added.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_INVALID_NVR
HCERR_ILLEGAL_ARGUMENT
```

Retrieving Name-Value Records

The following functions are defined to retrieve name-value records:

hc_nvr_get_count

Retrieves the number of metadata name and value tuples in this name-value record.

Synopsis

```
hcerr_t hc_nvr_get_count(hc_nvr_t *nvr,
     hc_long_t *retcount);
```

Description

This function retrieves the number of metadata name and value tuples in the specified name-value record.

Parameters

`nvr`

IN: Points to a name-value-record.

`retcount`

OUT: Updated to contain the count of name-value pairs.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_INVALID_NVR
```

hc_nvr_get_value_at_index

Iterates through the names and values in a name-value record.

Synopsis

```
hc_nvr_get_value_at_index(hc_nvr_t *nvr,
     hc_long_t index,
     char **namep,
     hc_value_t *valuep);
```

Description

This function iterates through the names and values in a name-value record. The returned names are read-only. Unpredictable results will occur if either the name or the value is referenced after either "hc_nvr_free" on page 66 or "hc_nvr_create_from_string_arrays" on page 84 is called on this name-value record.

Parameters

`nvr`

Points to a name-value-record.

`index`

IN: The index to examine.

`namep`

OUT: Updated to point to the attribute name at the specified index.

`valuep`

OUT: Updated with the "hc_value_t" on page 47 type-tagged value at the specified index.

Return Codes

```
HCERR_OK
HCERR_INVALID_NVR
HCERR_ILLEGAL_ARGUMENT
HCERR_NO_MORE_ARGUMENTS
HCERR_OOM
```

hc_nvr_get_long

Retrieves a value of type hc_long_t.

Synopsis

```
hcerr_t hc_nvr_get_long(hc_nvr_t *nvr,
     char *name,
     hc_long_t *retlong);
```

Description

This function retrieves the value of type hc_long_t (see "hc_type_t" on page 47) associated with an indicated attribute name in a name-value record.

Parameters

nvr

Points to a name-value-record.

name

IN: Attribute name to look for.

retlong

OUT: Updated to contain the hc_long_t value.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_INVALID_NVR
HCERR_ILLEGAL_VALUE_FOR_METADATA
HCERR_ILLEGAL_ARGUMENT
```

hc_nvr_get_double

Retrieves a value of type hc_double_t.

Synopsis

```
hcerr_t hc_nvr_get_double(hc_nvr_t *nvr,
     char *name,
     hc_double_t *retdouble);
```

Description

This function retrieves the value of type hc_double_t (see "hc_type_t" on page 47) associated with an indicated attribute name in a name-value record.

Parameters

nvr

Points to a name-value-record.

name

IN: Attribute name to look for.

retdouble

OUT: Updated to contain the hc_double_t value.

Return Codes

HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_INVALID_NVR
HCERR_ILLEGAL_VALUE_FOR_METADATA
HCERR_ILLEGAL_ARGUMENT

hc_nvr_get_string

Retrieves a value of a Unicode UTF-8 string.

Synopsis

```
hcerr_t hc_nvr_get_string(hc_nvr_t *nvr,
     char *name,
     hc_string_t *retstring);
```

Description

This function retrieves the value of a Unicode UTF-8 string associated with an indicated attribute name in a name-value record. Note that the memory pointed to will be freed when the record is freed.

Parameters

`nvr`

Points to a name-value-record.

`name`

IN: Attribute name to look for.

`retstring`

OUT: Updated to contain the `hc_string_t` value.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_INVALID_NVR
HCERR_ILLEGAL_VALUE_FOR_METADATA
HCERR_ILLEGAL_ARGUMENT
```

hc_nvr_get_binary

Retrieves a metadata value of type `binary`.

Synopsis

```
hcerr_t hc_nvr_get_binary(hc_nvr_t *nvr,
     hc_string_t name,
     int *size,
     unsigned char **bytes);
```

This function retrieves the value of type `binary` associated with an indicated attribute name in a name-value record. The binary data is not copied and is freed when the name-value record is freed.

Parameters

`nvr`

Points to a name-value-record.

`name`

IN: Name for the tuple.

`size`

OUT: Updated with the size of the data.

`bytes`

OUT: Updated to point to the binary data.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_INVALID_NVR
HCERR_ILLEGAL_ARGUMENT
HCERR_ILLEGAL_VALUE_FOR_METADATA
HCERR_NO_SUCH_ATTRIBUTE
```

hc_nvr_get_date

Retrieves metadata value of type `date`.

Synopsis

```
#include <time.h>
hcerr_t hc_nvr_get_date(hc_nvr_t *nvr,
     hc_string_t name,
     struct tm *value);
```

Description

This function retrieves the value of type `struct tm` associated with an indicated attribute name in a name-value record.

Parameters

`nvr`

Points to a name-value-record.

`name`

IN: Name for the tuple.

`value`

OUT: Updated with the `struct tm` (`time.h`) value.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_INVALID_NVR
HCERR_ILLEGAL_ARGUMENT
HCERR_ILLEGAL_VALUE_FOR_METADATA
HCERR_NO_SUCH_ATTRIBUTE
```

hc_nvr_get_time

Retrieves metadata value of type `time`.

Synopsis

```
#include <time.h>
hcerr_t hc_nvr_get_time(hc_nvr_t *nvr,
     hc_string_t name,
     time_t *value);
```

This function retrieves the value of type `time_t` (seconds since midnight) associated with an indicated attribute name in a name-value record.

Parameters

`nvr`

Points to a name-value-record.

`name`

IN: Name for the tuple.

`value`

OUT: Updated with the `time_t` (`time.h`) value.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_INVALID_NVR
HCERR_ILLEGAL_ARGUMENT
HCERR_ILLEGAL_VALUE_FOR_METADATA
HCERR_NO_SUCH_ATTRIBUTE
```

hc_nvr_get_timestamp

Retrieves metadata value of type `timestamp`.

Synopsis

```
#include <time.h>
hcerr_t hc_nvr_get_timestamp(hc_nvr_t *nvr,
     hc_string_t name,
     struct timespec *value);
```

This function retrieves the value of type `struct timespec` associated with an indicated attribute name in a name-value record.

Parameters

nvr

Points to a name-value-record.

name

IN: Name for the tuple.

value

OUT: Updated with the `struct timespec` (`time.h`) value.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_INVALID_NVR
HCERR_ILLEGAL_ARGUMENT
HCERR_ILLEGAL_VALUE_FOR_METADATA
HCERR_NO_SUCH_ATTRIBUTE
```

Creating and Converting Name-Value Records From and To String Arrays

The following functions are defined to create name-value records from string arrays and convert name-value records to string arrays:

- "hc_nvr_create_from_string_arrays" on page 84
- "hc_nvr_convert_to_string_arrays" on page 86

hc_nvr_create_from_string_arrays

Creates name-value-record from string names and string values.

Synopsis

```
hcerr_t hc_nvr_create_from_string_arrays(hc_session_t *session,
     hc_nvr_t **nvrp,
```

```
    char **names,
    char **values,
    hc_long_t nitems);
```

Description

This function creates a name-value-record from parallel tables of string names and string values. The correct metadata type for each name must be looked up from the schema associated with this session. The name-value record will automatically expand as needed. The names and data values are copied into the "hc_nvr_t" on page 48 structure, so the original names table and values table are left unchanged.

Note – Any time there is a conversion from a `double` type to or from a `string` type, there might be a loss of precision.

Parameters

`session`

IN: The session with which this name-value record is associated.

`nvrp`

OUT: Updated to point to a name-value-record.

`names`

IN: Points to an array of string names.

`values`

IN: Points to an array of string values.

`nitems`

IN: Number of active elements in the paired arrays.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_NULL_SESSION
HCERR_INVALID_SESSION
HCERR_ILLEGAL_ARGUMENT
```

hc_nvr_convert_to_string_arrays

Converts name-value-record to string names and string values.

Synopsis

```
hcerr_t hc_nvr_convert_to_string_arrays(hc_nvr_t *nvr,
     char ***namesp,
     char ***valuesp,
     int *nitemsp);
```

Description

This function converts a name-value-record into parallel tables of string names and string values. This destructively modifies the name-value record and frees it, so do not call "hc_nvr_free" on page 66 after calling this function.

When the conversion is finished, each string in the names and values tables should be freed with the designated deallocator (for example, `free`), as well as the names and values tables themselves.

Note – Any time there is a conversion from a `double` type to or from a `string` type, there might be a loss of precision.

Parameters

`nvr`

IN: The name-value-record.

`namesp`

OUT: Updated to point to an array of string names.

`valuesp`

OUT: Updated to point to an array of string values.

`nitemsp`

OUT: Updated to the number of active elements in the paired arrays.

Return Codes

`HCERR_OK`
`HCERR_BAD_REQUEST`

```
HCERR_OOM
HCERR_INVALID_NVR
HCERR_ILLEGAL_ARGUMENT
```

Storing Data and Metadata

The following functions are defined to store data and metadata and to enforce indexing of metadata where necessary:

- "hc_store_both_ez" on page 87
- "hc_store_metadata_ez" on page 88
- "hc_check_indexed_ez" on page 89

Note – The `is_indexed` value in the returned system record (`hc_system_record_t`) indicates whether this record was successfully inserted in the query engine at the time of store, and is hence available for query. Objects that were not inserted into the query engine at time of store are called *store index exceptions*. Until they are resolved, store index exceptions may or may not show up in the result sets of queries that match the store. A store index exception is resolved once the metadata for that object has been successfully inserted into the query engine, after which the object will definitely show up in the result sets of queries that match the store. The "hc_check_indexed_ez" on page 89 method may be used to attempt to resolve a store index exception under program control. Store index exceptions will also be resolved automatically (eventually) by ongoing system healing.

---

hc_store_both_ez

Stores a metadata record and associated data.

Synopsis

```
hcerr_t hc_store_both_ez (hc_session_t *session,
     *data_source_reader,
     void *cookie,
     hc_nvr_t *nvr,
     hc_system_record_t *system_record);
```

Description

This function stores both object data and metadata and returns a `system_record` descriptor. The status from this operation can be reclaimed using "hc_session_get_status" on page 55.

Parameters

`session`

IN: The session for the host and port to talk to.

`data_source_reader`

IN: The source of data to be stored. See "`read_from_data_source`" on page 49.

`cookie`

IN: An opaque data structure (cookie) to be provided to `data_source_reader`. For example, this could be a file descriptor.

`nvr`

IN: Pointer to a name-value record with the metadata.

`system_record`

OUT: Returned descriptor of a stored metadata record.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_NULL_SESSION
HCERR_INVALID_SESSION
HCERR_INVALID_NVR
HCERR_ILLEGAL_ARGUMENT
HCERR_NO_SUCH_TYPE
HCERR_XML_BUFFER_OVERFLOW
```

hc_store_metadata_ez

Adds a metadata record for the specified OID.

Synopsis

```
hcerr_t hc_store_metadata_ez(hc_session_t *session,
     hc_oid *oid,
     hc_nvr_t *nvr,
     hc_system_record_t *system_record);
```

Description

This function adds a metadata record for the specified OID and returns a `system_record` descriptor.

Parameters

`session`

IN: The session for the host and port to talk to.

`oid`

IN: An identifier of object data to which the metadata record is attached.

`nvr`

IN: Pointer to a name-value record with the metadata.

`system_record`

OUT: Returned descriptor of a stored metadata record.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_NULL_SESSION
HCERR_INVALID_SESSION
HCERR_INVALID_NVR
HCERR_INVALID_OID
HCERR_ILLEGAL_ARGUMENT
HCERR_XML_BUFFER_OVERFLOW
```

hc_check_indexed_ez

Checks if the metadata for an object is present in the query engine, and inserts it if not.

Synopsis

```
hcerr_t hc_check_indexed_ez(hc_session_t *session,
     hc_oid *oid,
     int *resultp);
```

Description

`checkIndexed` is intended as way to resolve a store index exception under program control (see "The 5800 System Query Integrity Model" on page 21). Once a store index exception occurs (as indicated by a non-zero value of the `is_indexed` field in the `hc_system_record_t` returned from a store operation) then `hc_check_indexed_ez` can be called repeatedly until it returns with `*resultp` set to any non-zero value. This will ensure that the metadata for the object has been inserted into the query engine; the object should then start to show up in matching queries.

Parameters

`session`

IN: The session for the host and port to talk to.

`oid`

IN: An identifier of object data to which the metadata record is attached.

`resultp`

OUT: Points to an `int` that is updated to a value that indicates if the metadata for this object has been inserted into the query engine. The returned value of `*resultp` is set to `-1` if the object was already present in the query engine, and is set to `0` if the object was not already in the query engine and could not be added, and to `1` if the object was just now added to the query engine. In other words, a non-zero value of `resultp` indicates that the store index exception has been resolved.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_NULL_SESSION
HCERR_INVALID_SESSION
HCERR_INVALID_OID
```

Retrieving Data and Metadata

The following functions are defined to retrieve data and metadata:

hc_retrieve_ez

Retrieves data for the specified OID.

Synopsis

```
hcerr_t hc_retrieve_ez(hc_session_t *session,
     *data_writer,
     void *cookie,
     hc_oid *oid);
```

Description

This function retrieves data for the specified OID.

Parameters

`session`

IN: The session for the host and port to talk to.

`data_writer`

IN: A function callback to store the retrieved data locally. See "write_to_data_destination" on page 50.

`cookie`

IN: The opaque data delivered to the `data_writer` callback to identify this data stream.

`oid`

IN: Identifier for the metadata record to retrieve.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
```

```
HCERR_OOM
HCERR_NULL_SESSION
HCERR_INVALID_SESSION
HCERR_INVALID_OID
```

hc_retrieve_metadata_ez

Retrieves a metadata record for the specified OID.

Synopsis

```
hcerr_t hc_retrieve_metadata_ez (hc_session_t *session,
     hc_oid *oid,
     hv_nvr_t **nvrp);
```

Description

This function retrieves a metadata record for the specified OID. When it has finished, you should call "hc_nvr_free" on page 66 to free the name-value-record.

Parameters

`session`

IN: The session for the host and port to talk to.

`oid`

IN: An identifier of the metadata record to retrieve.

`nvrp`

OUT: Updated with a pointer to a dynamically allocated name-value record with the metadata.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_NULL_SESSION
HCERR_INVALID_SESSION
HCERR_INVALID_OID
```

hc_range_retrieve_ez

Retrieves a specified range of data for a specified OID.

Synopsis

```
hc_range_retrieve_ez(hc_session_t *session,
     write_to_data_destination data_writer,
     void *cookie,
     hc_oid *oid,
     hc_long_t; firstbyte,
     hc_long_t lastbyte);
```

Description

This function retrieves a specified range of data for a specified OID.

Parameters

`session`

IN: The session for the host and port to talk to.

`data_writer`

IN: Function callback to store the retrieved data locally.

`cookie`

IN: Opaque data delivered to the `data_writer` callback to identify this data cookie.

`oid`

IN: An identifier of the data record to retrieve.

`firstbyte`

IN: First byte of data range to retrieve.

`lastbyte`

IN: Last byte of data range to retrieve, or `-1` for the end of the record.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
```

```
HCERR_NULL_SESSION
HCERR_INVALID_SESSION
HCERR_INVALID_OID
HCERR_ILLEGAL_ARGUMENT
```

Querying Metadata

The following functions are defined for simple queries:

The following functions are defined for prepared statement queries:

Prepared statement example:

hc_query_ez

Retrieves OIDs and optionally name-value records matching a query.

Synopsis

```
hcerr_t hc_query_ez(hc_session_t *session,
     hc_string_t query,
     hc_string_t selects[],
     int n_selects,
```

```
    int results_per_fetch,
    hc_query_result_set_t **rsetp);
```

Description

This function retrieves OIDs and optionally name-value records matching a query. If the `selects` list is `NULL`, only OIDs are retrieved. If `selects` is not `NULL`, name-value records are also retrieved and should each be freed using "`hc_nvr_free`" on page 66. In both cases the result set should be freed using "`hc_qrs_free`" on page 99.

Note – When a query is incorrect and elicits an error from the server, the error is often reported after "`hc_qrs_free`" on page 99and not from `hc_query_ez`. Your application should be prepared to receive and report an error from either place.

Parameters

`session`

IN: The session for the host and port to talk to.

`query`

IN: Query (`where` clause with names in single quotes).

`selects`

IN: Points to an array of "`hc_string_t`" on page 46, each member of which is the name of a field to retrieve from the metadata (`select` clause). Set to `NULL` to only retrieve OIDs matching the query.

`n_selects`

IN: The number of items in the `select` clause.

`results_per_fetch`

IN: The number of results to return on each fetch from the server. `results_per_fetch` must be greater than `0`.

`rsetp`

OUT: Updated to point to the new result set. See "`hc_query_result_set_t`" on page 49.

Return Codes

```
HCERR_OK
HCERR_OOM
HCERR_BAD_REQUEST
HCERR_NULL_SESSION
HCERR_INVALID_SESSION
HCERR_ILLEGAL_ARGUMENT
```

See Also

"hc_qrs_free" on page 99

hc_qrs_next_ez

Fetches the next OID and optionally name-value record from the QueryResultSet.

Synopsis

```
hcerr_t hc_qrs_next_ez(**rset,
     hc_oid *oid,
     hc_nvr_t **nvrp,
     int *finishedp);
```

Description

This function fetches an OID and optionally name-value record from the query result set. Once the last result is fetched, in subsequent calls the int pointed to by finishedp is set to 1.

Parameters

rset

IN: Current query result set. See "hc_query_result_set_t" on page 49.

oid

OUT: Points to an OID that is updated to the OID of a record matching the query, assuming finishedp is 0.

nvrp

OUT: Updated to point to a name-value record with the metadata from the OID matching the query, assuming the query specified selects and assuming finishedp is 0. Note that you must free the name-value record using "hc_nvr_free" on page 66.

`finishedp`

OUT: Points to an `int` that is updated to `0` if query data has been returned and to `1` if the result set is empty.

Return Codes

```
HCERR_OK
HCERR_OOM
HCERR_BAD_REQUEST
HCERR_INVALID_RESULT_SET
HCERR_ILLEGAL_ARGUMENT
```

hc_qrs_is_query_complete

Indicates whether results of this query are complete in the sense that all objects that match the query, aside from possible store index exceptions, are included in the result set,

Synopsis

```
hcerr_t hc_qrs_is_query_complete(hc_query_result_set_t *rset,
     int *completep);
```

Description

Indicates whether results of this query are complete in the sense that all objects that match the query, aside from possible store index exceptions, are included in the result set. Applications that depend on completeness of query results can interrogate `hc_qrs_is_query_complete` after retrieving all the query results that match a particular query. When `completep` is set to `1`, the only items that should be missing from the result set are store index exceptions that were indicated to the application by a value of `0` in the `is_indexed` field of the `hc_system_record_t` structure returned from the store.

Parameters

`rset`

IN: Current query result set. See "`hc_query_result_set_t`" on page 49.

`completep`

OUT: Points to an `int` that is updated to `1` if all objects that match the query (other than potential store index exceptions) should be present in the result set

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_INVALID_RESULT_SET
```

hc_qrs_get_query_integrity_time

Returns a time that helps get more detail on which store index exceptions might still be unresolved.

Synopsis

```
hcerr_t hc_qrs_get_query_integrity_time(hc_query_result_set_t *rset,
     hc_long_t *query_timep);
```

Description

If the query integrity time is non-zero, then all store index exceptions whose object creation time falls before the query integrity time have been resolved. Stored objects from before that time should show up in all matching query result sets. Store index exceptions that occurred after that time may not yet have been resolved, and hence might still be missing from a matching query result set. If the Query Integrity Time is zero, then the set of results in this `ResultSet` is not known to be complete. Note that `hc_is_query_complete` will return a non-zero `completep` value if and only if `hc_get_query_integrity_time` would set `query_timep` to non-zero query integrity time.

Time values from `getQueryIntegrityTime` can be compared to object creation time values returned in the `creation_time` field of the `hc_system_record_t` structure to determine if a particular store operation has been resolved. Note: the query integrity time as reported may well be earlier than the actual oldest time of a still-unresolved store index exception. The query integrity time can even go backwards, in other words, a later query can report an earlier query integrity time.

Parameter

`rset`

Updated to point to the new query result set. See "`hc_query_result_set_t`" on page 49.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
```

```
HCERR_OOM
HCERR_INVALID_RESULT_SET
```

hc_qrs_free

Releases the resources associated with this `QueryResultSet`.

Synopsis

```
hcerr_t hc_qrs_free (**rsetp);
```

Description

This function releases the resources associated with this `QueryResultSet`.

Note – When a query is incorrect and elicits an error from the server, the error is often reported after `hc_qrs_free` and not from "hc_query_ez" on page 94. Your application should be prepared to receive and report an error from either place.

Parameter

`rset`

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_INVALID_RESULT_SET
```

See Also

"hc_query_ez" on page 94

"hc_pstmt_query_ez" on page 109

hc_pstmt_create

Creates an "hc_pstmt_t" on page 49 for use with the "hc_pstmt_query_ez" on page 109 function.

Synopsis

```
hcerr_t hc_pstmt_create(hc_session_t *session,
     hc_string_t query,
     hc_pstmt_t **ptr);
```

Description

This function creates a prepared statement for use with the "hc_pstmt_query_ez" on page 109 function.

Parameters

`session`

IN: `session` that this query will be used with.

`query`

IN: Query (`where` clause with "?' for values).

`ptr`

OUT: Updated to point to opaque "hc_pstmt_t" on page 49.

Return Codes

```
HCERR_OK
HCERR_OOM
```

See Also

"hc_pstmt_query_ez" on page 109

hc_pstmt_free

Frees a "hc_pstmt_t" on page 49 with all its bindings.

Synopsis

```
hcerr_t hc_pstmt_free(hc_pstmt_t *pstmt);
```

Description

This function frees a prepared statement.

Parameters

`pstmt`

Prepared statement to be freed.

Return Codes

`HCERR_OK`

See Also

"hc_pstmt_create" on page 100

hc_pstmt_set_string

Adds a string binding to a "hc_pstmt_t" on page 49.

Synopsis

```
hcerr_t hc_pstmt_set_string(hc_pstmt_t *pstmt,
     int which,
     hc_string_t value);
```

Description

This function binds an "hc_string_t" on page 46 to one of the variables in a prepared statement. The variable must be of the appropriate type in the database, that is, string (UTF-8). Errors in binding and type are returned when the "hc_pstmt_t" on page 49 is used to query the server.

Parameters

`pstmt`

Prepared statement to add the binding to.

`which`

IN: Variable (”?’) in the prepared statement, numbered from `1`.

`value`

IN: String to bind.

Return Codes

```
HCERR_OK
HCERR_OOM
```

See Also

“hc_pstmt_create” on page 100

hc_pstmt_set_char

Adds a char binding to a “hc_pstmt_t” on page 49.

Synopsis

```
hcerr_t hc_pstmt_set_char(hc_pstmt_t *pstmt,
     int which,char *value);
```

Description

This function binds a `char *` Latin-1 string to one of the variables in a prepared statement. The variable must be of the appropriate type in the database. Errors in binding and type are returned when the “hc_pstmt_t” on page 49 is used to query the server.

Parameters

`pstmt`

Prepared statement to add the binding to.

`which`

IN: Variable (”?’) in the prepared statement, numbered from 1.

`value`

IN: `char *` string to bind.

Return Codes

```
HCERR_OK
HCERR_OOM
```

See Also

“hc_pstmt_create” on page 100

hc_pstmt_set_double

Adds a double precision binding to a “hc_pstmt_t” on page 49.

Synopsis

```
hcerr_t hc_pstmt_set_double(hc_pstmt_t *pstmt,
     int which,
     hc_double_t value)
```

Description

This function binds an “hc_double_t” on page 46 to one of the variables in a prepared statement. The variable must be of the appropriate type in the database. Errors in binding and type are returned when the “hc_pstmt_t” on page 49 is used to query the server.

Parameters

`pstmt`

Prepared statement to add the binding to.

`which`

IN: Variable (”?’) in the prepared statement, numbered from 1.

`value`

IN: Double precision value to bind.

Return Codes

HCERR_OK
HCERR_OOM

See Also

"hc_pstmt_create" on page 100

hc_pstmt_set_long

Adds a "hc_long_t" on page 46 binding to a "hc_pstmt_t" on page 49.

Synopsis

```
hcerr_t hc_pstmt_set_long(hc_pstmt_t *pstmt,
     int which,
     hc_long_t value);
```

Description

This function binds an "hc_long_t" on page 46 to one of the variables in a prepared statement. The variable must be of the appropriate type in the database. Errors in binding and type are returned when the "hc_pstmt_t" on page 49 is used to query the server.

Parameters

pstmt

Prepared statement to add the binding to.

which

IN: Variable ("?') in the prepared statement, numbered from 1.

value

IN: "hc_long_t" on page 46 value to bind to.

Return Codes

HCERR_OK
HCERR_OOM

See Also

"hc_pstmt_create" on page 100

hc_pstmt_set_date

Adds a date binding to a "hc_pstmt_t" on page 49.

Synopsis

```
#include <time.h>
hcerr_t hc_pstmt_set_date(hc_pstmt_t *pstmt,
     int which,
     struct tm *value);
```

Description

This function binds a date in the form of the POSIX `struct` to one of the variables in a prepared statement. The variable must be of the appropriate type in the database. Errors in binding and type are returned when the "hc_pstmt_t" on page 49 is used to query the server.

The `struct tm` fields are as defined in the POSIX standard and interpreted by `mktime(3C)`. All fields are ignored except:

```
int tm_mday; /* day of the month - [1, 31] */
int tm_mon; /* months since January - [0, 11] */
int tm_year; /* years since 1900 */
```

Parameters

`pstmt`

Prepared statement to add the binding to.

`which`

IN: Variable ("?') in the prepared statement, numbered from `1`.

`value`

IN: `struct tm` (`time.h`) value to bind.

Return Codes

`HCERR_OK`
`HCERR_OOM`

See Also

"hc_pstmt_create" on page 100

hc_pstmt_set_time

Adds a time-of-day binding to a "hc_pstmt_t" on page 49.

Synopsis

```
#include <time.h>
hcerr_t hc_pstmt_set_time(hc_pstmt_t *pstmt,
     int which,
     time_t *value);
```

Description

This function binds a time of day in seconds to one of the variables in a prepared statement. The variable must be of the appropriate type in the database. Errors in binding and type are returned when the "hc_pstmt_t" on page 49 is used to query the server.

Parameters

`pstmt`

Prepared statement to add the binding to.

`which`

IN: Variable ("?') in the prepared statement, numbered from `1`.

`value`

IN: `time_t` (`time.h`) value to bind.

Return Codes

```
HCERR_OK
HCERR_OOM
```

See Also

"hc_pstmt_create" on page 100

hc_pstmt_set_timestamp

Adds a timestamp binding to a "hc_pstmt_t" on page 49.

Synopsis

```
#include <time.h>
hcerr_t hc_pstmt_set_timestamp(hc_pstmt_t *pstmt,
     int which,
     struct timespec *value);
```

Description

This function binds a timestamp in the form of the POSIX `struct timespec` to one of the variables in a prepared statement. The variable must be of the appropriate type in the database. Errors in binding and type are returned when the "hc_pstmt_t" on page 49 is used to query the server.

Parameters

`pstmt`

Prepared statement to add the binding to.

`which`

IN: Variable ("?') in the prepared statement, numbered from `1`.

`value`

IN: `struct timespec` (`time.h`) value to bind.

Return Codes

```
HCERR_OK
HCERR_OOM
```

See Also

"hc_pstmt_create" on page 100

hc_pstmt_set_binary

Adds a binary binding to a "hc_pstmt_t" on page 49.

Synopsis

```
hcerr_t hc_pstmt_set_binary(hc_pstmt_t *pstmt,
     int which,
     unsigned char *data,int size);
```

Description

This function binds a binary array to one of the variables in a prepared statement. The variable must be of the appropriate type in the database. Errors in binding and type are returned when the "hc_pstmt_t" on page 49 is used to query the server.

Parameters

`pstmt`

Prepared statement to add the binding to.

`which`

IN: Variable ("?') in the prepared statement, numbered from `1`.

`data`

IN: Pointer to binary data to bind.

`size`

IN: Number of bytes in array of binary data.

Return Codes

```
HCERR_OK
HCERR_OOM
```

See Also

"hc_pstmt_create" on page 100

hc_pstmt_query_ez

Retrieves OIDs and optionally name-value records matching a prepared statement.

Synopsis

```
hcerr_t hc_pstmt_query_ez(*pstmt,hc_string_t selects[],
     int n_selects,
     int results_per_fetch,
     hc_query_result_set_t **rsetp);
```

Description

This function retrieves OIDs and optionally name-value records matching a prepared statement. "hc_qrs_next_ez" on page 96 is used to access the results in the result set. If the `selects` list is `NULL`, only OIDs are retrieved. If `selects` is not `NULL`, name-value records are also retrieved and should each be freed using "hc_nvr_free" on page 66. In both cases the result set should be freed using "hc_qrs_free" on page 99.

Note – When a query is incorrect and elicits an error from the server, the error is often reported after "hc_qrs_free" on page 99 and not from "hc_pstmt_query_ez" on page 109. Your application should be prepared to receive and report an error from either place.

Parameters

`pstmt`

IN: Prepared statement generated by "hc_pstmt_create" on page 100.

`selects`

IN: Points to an array of "hc_string_t" on page 46, each of which is the name of a field to retrieve from the metadata (`select` clause). Set to `NULL` to only retrieve OIDs matching the query.

`n_selects`

IN: The number of items in the select clause.

`results_per_fetch`

IN: The number of results per internal fetch.

`rsetp`

OUT: Updated to point to the new result set. See "hc_query_result_set_t" on page 49.

Return Codes

```
    HCERR_OK
    HCERR_OOM
    HCERR_BAD_REQUEST
    HCERR_NULL_SESSION
    HCERR_INVALID_SESSION
HCERR_ILLEGAL_ARGUMENT
```

See Also

"hc_pstmt_create" on page 100

Querying With a Prepared Statement

The following code is an example of querying with a prepared statement. Error handling is omitted. Two metadata fields are used with the definitions from the schema:

```
    <field name="test_date" type="date">
    <field name="test_status" type="string">

    hcerr_t res;
    time_t t;
    struct tm *date;
    hc_pstmt_t *pstmt;
    hc_query_result_set_t *rset;
    hc_string_t selects[] = { "test_status" };

    // get today's date
    time(&t);
    date = gmtime(&t);

    // list all OIDs with today's date
    res = hc_pstmt_create(session, "test_date = ?", &pstmt);
    res = hc_pstmt_set_date(pstmt, 1, date);
    res = hc_pstmt_query_ez(pstmt, NULL, 0, 2000, &rset);

    while (1) {
        hc_oid oid;
        int finished;
        res = hc_qrs_next_ez(rset, &oid, NULL, &finished);
        if (finished)
            break;
        printf("today's oid: %s\n", oid);
    }
    res = hc_qrs_free(rset);
```

```
// list all OIDs from yesterday with test_status
t = 86400; // 86400 sec/day
date = gmtime(&t);

res = hc_pstmt_set_date(pstmt, 1, date);
res = hc_pstmt_query_ez(pstmt, selects, 1, 2000, &rset);

while (1) {
    hc_oid oid;
    hc_nvr_t *nvr         int finished;
    hc_string_t test_status;

    res = hc_qrs_next_ez(rset, &oid, &nvr, &finished);
    if (finished)
        break;

    res = hc_nvr_get_string(nvr, "test_status", &test_status);
    printf("yesterday's oid & test_status: %s %s\n", oid, test_status);
    hc_nvr_free(nvr);
}
res = hc_qrs_free(rset);
```

Deleting Records

The following function is defined to delete records: "hc_delete_ez" on page 111.

hc_delete_ez

Deletes the metadata record for specified OID.

Synopsis

```
hcerr_t hc_delete_ez(hc_session_t *session,
     hc_oid *oid);
```

Description

This function deletes the metadata record for the specified OID. When the last metadata record associated with a data object is deleted, the underlying data object is also deleted.

Parameters

`session`

IN: Pointer to the session.

`oid`

IN: The specified OID.

Return Codes

```
HCERR_OK
HCERR_BAD_REQUEST
HCERR_OOM
HCERR_NULL_SESSION
HCERR_INVALID_SESSION
HCERR_INVALID_OID
```

Translating Error and Type Codes

The following functions are defined for translating error codes and type codes into strings:

- "hc_decode_hcerr" on page 112
- "hc_decode_hc_type" on page 113

hc_decode_hcerr

Translates an error code into a string.

Synopsis

```
char *hc_decode_hcerr(hcerr_t res);
```

Description

Translates an error code into a string.

Parameter

`res`

IN: The error code returned by a function.

hc_decode_hc_type

Translates a type code into a string.

Synopsis

```
char *hc_decode_hc_type(hc_type_t type);
```

Description

Translates a type code into a string.

Parameters

`type`

IN: The type code to translate.

◆ ◆ ◆ CHAPTER 4

Sun StorageTek 5800 System Query Language

This chapter provides information on the 5800 system query language.

Note – For details of the metadata system and how it is configured, see Chapter 8, "Configuring Metadata and Virtual File System Views" in *Sun StorageTek 5800 Storage System Administration Guide*.

The following topics are discussed:

Interfaces

The 5800 system Java and C APIs both have a `query` function that passes a query string to the 5800 system. Queries in the 5800 system are presented to the name-value metadata cache.

Operation

The query format is similar to the `where` clause of an SQL query. The two main differences are that 5800 system queries do not contain embedded subqueries, and that the only "columns" that are available are the attributes defined in the 5800 system schema.

Many features of the underlying metadata database's own query language can be used in queries. There is a recommended subset of queries, however, that is most likely to be portable from the 5800 system emulator to a live 5800 system cluster. That subset is described in the sections "Supported Expression Types" on page 121 and "Queries Not Supported in Version 1.1" on page 123. These are the query expression types that should work identically on the 5800 system emulator and a live 5800 system cluster.

Supported Data Types

- Long— 8-byte integer value.
- Double— 8-byte IEEE 754 double-precision floating point value.
- String— now allows all Unicode values from the Basic Multilingual Plane (BMP). The encoding used is UTF-8. the schema definition of each String attribute must specify a length. `String(N)` is used as the convention to refer to the type of a String attribute whose length is set to `N`.
- char— similar to String, except that it is limited to 8-bit characters in the ISO-8859-1 (Latin-1) character set.
- Date— corresponds to the JDBC SQL DATE type. Year/Month/Day.
- Time— corresponds to the JDBC SQL TIME type. The Java `java.sql.Time` type only allows specification of whole seconds.
- Timestamp— corresponds to the JDBC SQL TIMESTAMP type with precision `3` (absolute Year/Month/Day/Hour/Minute/Second/Millisecond).
- Binary— string of binary bytes.
- Objectid— similar to binary, with internal support for sub-fields. Reserved for use by the `system.object_id` field. Other fields that must store an OID should use the string or binary type for that field.

Queries

A query in the 5800 system query language is translated into an equivalent query for the underlying database that implements the query engine. The database used in a live 5800 system is Sun's High Availability Database (HADB). The database used by the 5800 system emulator is Apache™ Derby. Since the SQL query language used by HADB and Derby differ in substantial ways, a subset of the query language is provided for portability between the cluster and the 5800 system emulator to enable application development in the emulator environment and subsequent deployment of the applications to a live 5800 system.

Translating a Query to the Underlying Database

The following provides a summary of the translation of the 5800 system queries to SQL queries that are presented to the underlying database.

The metadata schema specifies the layout of fields into tables and columns. When the schema is committed, a particular set of actual tables and columns is created in the underlying database that matches the format of the table layout in the schema.

When translating a 5800 system query to SQL, each field name in the query is translated into a reference to the particular column and particular table that represents that field. Typed literal values are translated into a form that the extended metadata cache knows how to deal with. Specifically, most literal values are replaced with an equivalent dynamic parameter. Thus, the list of dynamic parameters that the underlying database uses combines both the dynamic parameters and also many of the literal values from the 5800 system query. Finally, an implicit `INNER JOIN` is introduced between all the tables containing the translated query fields. Everything else (usually database expression syntax) is left unchanged, allowing almost all the database engine's powerful query syntax to be used with 5800 system queries.

The presence of the `INNER JOIN` has important consequences when queries are evaluated. An object is only returned by a query when all of the fields referenced by the query itself and all of the fields referenced in the select list of the query all have non-null values. Queries with OR clauses, in particular, can produce non-intuitive results. As an extreme example, consider a query: "fieldA is not null OR fieldB is not null." This query will not select an object unless both fieldA and fieldB are non-null, because of this implicit inner join.

Attribute Format in Queries

Any string in double quotes (for example, "filename") and any dotted string in Java Identifier format (for example, `mp3.title`) will automatically be treated as an attribute name. The double quotes can optionally be omitted even on a non-dotted name as long as the attribute does not match an SQL reserved word in any of the Sun StorageTek 5800 underlying metadata databases.

Attribute names must appear in the current 5800 system schema to be used in a query. This is because the proper type information about each attribute must be derived to build the query.

SQL Syntax in 5800 System Queries

General Unicode characters outside of the ASCII range in queries are allowed in only two places to the 5800 system. Specifically, both attribute names and literal values may contain general Unicode characters. All text that is not either an attribute name nor a literal value is passed unchanged to the underlying query engine, and must consist of ASCII characters only. An attempt to pass non-ASCII characters in a query will result in an error.

Literals In Queries

This section details the kinds of literals that can occur in 5800 system queries.

Dynamic Parameters

5800 system queries allow dynamic parameters. A dynamic parameter is indicated by the presence of a question mark in the query string (for example, the query `name=? AND address=?`). The `bindParameter` call is used in Java to bind typed values for use in place of the question marks.

For the Java API, the syntax is:

```
import com.sun.honeycomb.client.PreparedStatement;
import com.sun.honeycomb.client.QueryResultSet;
Date date_value= new Date();
PreparedStatement stmt = new PreparedStatement(
"system.test.type_date=?");
stmt.bindParameter(date_value,1);
QueryResultSet qrs = query(stmt);
```

For further information, see “`query` (with `PreparedStatement`)” on page 35, “`query` (with `PreparedStatement` and `selectKeys`)” on page 35 and “`PreparedStatement`” on page 36.

String Literals

String literals are surrounded by single quotes (for example, 'The Lighter Side'). You can embed single quote characters in a query by doubling them (for example, `'Susan''s House'`). Any UTF-8 string can be included in a string literal (except the null character, which is treated as a string terminator by the C API).

Numeric Literals

Only ASCII digits are recognized as numeric literals. For example, `45`, `-1`, `3.14`, `5.2E10`. Digits from other parts of the Unicode code space will cause a parse error.

Literals for 5800 System Data Types

For each 5800 system data type, there is a syntax to include literals of that type in a query string. The syntax is `{type_name 'stringliteral'}`. For example, consider the query:

```
timestamp_field<{timestamp '2006-10-26T12:00:00Z'}
```

In particular, this syntax can be used to query for a particular object ID:

```
system.object_id = {objectid
'0200011e61c159bdfa654e11db8a45cafecafecafe00000000200000000'}
```

For comparing against binary values, either of the following forms may be used:

```
binary_field = x'beeffeed'
```

```
binary_field = {binary 'beeffeed'}
```

For more information, see "Canonical String Format" on page 119.

Canonical String Format

Each type in a 5800 system has a canonical representation as a `string` value. The canonical `string` representation of each type is shown in Table 4–1.

TABLE 4–1 Canonical String Representation of Data Types

Data Type	Canonical String Representation
`STRING`	The string itself.
`CHAR`	The string itself.
`BINARY`	Hexadecimal dump of the value with two hex digits per byte.
`LONG`	Result of `Long.toString`. For example, `88991` or `-7975432785`.
`DOUBLE`	Result of `Double.toString`. For example, `1.45` or `NaN` or `-Infinity` or `-1.56E200`.
`DATE`	`YYYY-mm-dd`. For example, `2001-01-01`.
`TIME`	`HH:mm:sss`. For example, `12:02:01`.
`TIMESTAMP`	`YYYY-mm-ddThh:mm:ss.fffZ` (time relative to UTC). For example, `1969-12-31T23:59:59.999Z`.
`OBJECTID`	60-digit hexadecimal dump of the `objectid`.

This canonical string encoding is used in the following places:

- When exposing the field as a directory component or a filename component in a virtual view

- When converting a typed value to a string as the result of the `getAsString` operation on a `NameValueRecord` or a `QueryResultSet` operation
- When parsing a literal value as described in "Literals for 5800 System Data Types" on page 119 to create a typed query value from a string representation of that value.

The Canonical String Decode Operation

The inverse of the canonical string encoding is used in the following places:

- It is always allowed to store a `string` value into any metadata field, no matter what the type of the field is. The actual data stored is the result of applying the canonical string decode operation to the incoming `string` value.
- On a virtual view lookup operation, the canonical string decode operation is used on the supplied filename to derive the actual metadata values to look up, given their string representations in the filename.

  The decode operation is allowed to accept incoming `string` values that would never be a legal output for an encode operation. Some examples of this include:

  – `decodeBinary` of an odd number of hex digits. The convention is to left-justify the supplied digits in the binary value. For example, the string `"b0a"` corresponds to the binary literal `[b0a0]`.

  – `decodeDate` of a non-standard date format.

  – A `double` value encoded with a non-canonical number of digits. For example, `.00145E20` instead of `1.45E17`.

EXAMPLE 4–1 Virtual View Lookup Operation

If you take a value `V` and encode it into a string `S`, and then perform the canonical decode operation on `S` to get a new value `V'`. Does `V` always equal `V'`? The answer is yes in most cases, but not always.

What is actually guaranteed is the weaker statement that if `encode(V) = S` and if `decode(S)=V'`, then `encode(V')` is also equal to `S`.

JDBC and HADB Date and Time Operations

- Use the 5800 system literal format for all the Date and Time operations, for example, `{date '12-31-2007'}`.
- The JDBC standard escape sequences for date (`{d 'YYYY-mm-dd'}`) and time (`{t 'hh:mm:ss'}`) literals are available. However, usage of JDBC format date and time literals may produce inconsistent results. In particular, when JDBC format is used, the literal format is interpreted as being relative to the local time zone, and the time zone usually differs between a 5800 system cluster and client machine.

- The following JDBC function escapes supported:

  `TIMESTAMPDIFF, TIMESTAMPADD, CURRENT_TIMESTAMP, CURRENT_TIME, CURRENT_DATE, HOUR, MINUTE, SECOND.`

Reserved Words

Some SQL reserved words (such as `BETWEEN` or `LIKE`) are allowed in queries and are expected to occur. An SQL reserved word cannot be used as an attribute name unless it is enclosed in double quotes (for example, `"FIRST"`). Some other SQL reserved words (such as `SELECT` or `CREATE`) are forbidden from occurring unquoted in queries. Any query containing these words unquoted will immediately return an error. These forbidden words can be used as attribute names by enclosing them in double quotes.

Supported Expression Types

The following expression types are explicitly supported in this release of the 5800 system:

- Make use of JDBC escape syntax wherever possible. The escape syntax makes the query syntax more portable without losing functionality.
- Comparison operations: `expr1 OP expr2`, where `OP` is one of `=`, `!=`, `<>`, `<=`, `>=`, `<`, or `>`.

  The comparison operations can compare any two expressions (for example, two attribute values or one attribute value and one literal value). The two values must be of compatible types. For example, a `<` comparison cannot be used between a numeric literal value and a string-valued attribute. Note that `<>` and `!=` are synonyms.
- The concatenation operator, `expr1 || expr2`.

  The concatenation operator `||` is an SQL standard way of concatenating two expressions to produce a combined output string.
- Use of parentheses to indicate precedence of evaluation.
- Boolean operators `AND`, `OR`, and `NOT`.
- `expr1 [NOT] LIKE expr2`.
  - Attempts to match a character expression to a character pattern, which is a character string that includes one or more wildcards.
  - `%` matches any number (zero or more) of characters in the corresponding position in `expr1`.
  - `_` matches one character in the corresponding position in `expr1`.
  - Any other character matches only that character in the corresponding position in the character expression.
- `expr1 BETWEEN expr2 AND expr3`

- `expr [NOT] IN (valueslist)`.

Note – The 5800 system emulator supports (but the cluster database does not) a JDBC "escape" clause that allows you to treat either `%` or `_` as constant characters. There is currently no way to accomplish this in a `LIKE` clause in a query on a live cluster.

- The following JDBC function escapes have been tested and are supported:
  - `{fn UCASE(string)}` to convert a string to uppercase.
  - `{fn LCASE(string)}` to convert a string to lowercase.
  - `{fn ABS(value)}` to take the absolute value of a numeric expression.
  - `{fn LENGTH(string)}` to get the length of a string.
  - `{fn SUBSTRING(string, start, length)}` to get a character string formed by extracting `length` characters from `string` beginning at `start`.
  - `{fn LOCATE(string1, string2[, start])}` to locate the position in `string2` of the first occurrence of `string1`, searching from the beginning of `string2`. If `start` is specified, the search begins from position `start`. `0` is returned if `string2` does not contain `string1`.
  - `{fn LTRIM(string)}` to remove the leading blank spaces in a character string
  - `{fn RTRIM(string)}` to remove the trailing blank spaces of a character string
  - `{fn CONCAT(string1, string2)}` to get the concatanated character string by appending `string2` to `string1`. If a string is `NULL`, the result is DBMS-dependent.
  - `{fn TIMESTAMPDIFF(interval, timestamp1, timestamp2)}`. An integer representing the number of `interval` by which `timestamp2` is greater than `timestamp1`. `interval` may be one of the following: `SQL_TSI_FRAC_SECOND`, `SQL_TSI_SECOND`, `SQL_TSI_MINUTE`, `SQL_TSI_HOUR`, `SQL_TSI_DAY`, `SQL_TSI_MONTH`, `SQL_TSI_QUARTER`, or `SQL_TSI_YEAR`.
  - `{fn TIMESTAMPADD(interval, count, timestamp)}`. A timestamp calculated by adding `count` number of `intervals` to `timestamp`. `interval` may be one of the following: `SQL_TSI_FRAC_SECOND`, `SQL_TSI_SECOND`, `SQL_TSI_MINUTE`, `SQL_TSI_HOUR`, `SQL_TSI_DAY`, `SQL_TSI_WEEK`, `SQL_TSI_MONTH`, `SQL_TSI_QUARTER`, or `SQL_TSI_YEAR`.
- `field_name IS NOT NULL` to return all objects that have a value for a specific field. In particular, the query "`system.object_id IS NOT NULL`" can be used to query all the objects in the system.

Note – The form "`field_name IS NULL`" to identify all the objects that lack a value for a specific field is not supported and will not work properly in all cases.

Examples of Supported Query Expressions

- `{fn LCASE(mp3.artist)} LIKE '%floyd%' AND system.object_size > 2000000`
- `(object_size < 200) OR "Collation" = 'en-US'`
- `{fn TIMESTAMPDIFF(SQL_TSI_YEAR, system.test.type_timestamp, '2007-04-02 01:50:50.999')} < 3`
- `{fn TIMESTAMPADD(SQL_TSI_YEAR, 2, system.test.type_timestamp )} > '2007-04-03 01:50:50.999'`

Queries Not Supported in Version 1.1

The following JDBC escapes are not supported:

- `CHAR(code)`
- `INSERT(string1, start, length, string2)`
- `LEFT(string, count)`
- `REPEAT(string, count)`
- `REPLACE(string1, string2, string3)`
- `RIGHT(string, count)`
- `SPACE(count)`
- `ROUND(value, number)`
- `TRUNCATE(value, number)`
- `POWER(value, power)`
- `ACOS(float)`
- `ASIN(float)`
- `ATAN(float)`
- `ATAN2(float1, float2)`
- `CEILING(number)`
- `COS(float)`
- `COT(float)`
- `DEGREES(number)`
- `EXP(float)`
- `FLOOR(number)`
- `LOG(float)`
- `LOG10(float)`
- `ASCII(string)`
- `MOD(float1, float2)`
- `PI()`
- `POWER(number, power)`
- `RADIANS(number)`
- `RAND(integer)`
- `SIGN(number)`

- SIN(float)
- SQRT(float)
- TAN(float)

SQL Words That Are Allowed in Queries

Some SQL reserved words (such as BETWEEN or LIKE) are allowed in queries and are expected to occur. An SQL reserved word cannot be used as an attribute name unless it is enclosed in double quotes (for example, "FIRST").

The following reserved words are allowed:

ABS, ABSOLUTE, AFTER, AND, AS, ASCII, AT, BEFORE, BETWEEN, BINARY, BIT, BIT_LENGTH, BOOLEAN, BOTH, CASE, CAST, CHAR, CHARACTER, CHARACTER_LENGTH, CHAR_LENGTH, COALESCE, COLLATE, COLLATION, CONTAINS,COUNT, CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, DATE, DAY, DEC,DECIMAL, DOUBLE, ELSE, ELSEIF, END, ESCAPE, EXISTS, EXIT, EXPAND,EXTRACT, FALSE, FIRST, FOR, FROM, HOUR, IF, IN, INT, INTEGER, INTERVAL,IS, LCASE, LEADING, LEFT, LIKE, LOWER, MATCH, MAX, MIN, MINUTE, MONTH,NCHAR, NO, NOT, NULL, NUMERIC, OCTET_LENGTH, OF, R, PAD, PI,POSITION, REAL, RIGHT, RTRIM, SECOND, SIZE, SMALLINT, SUBSTRING, THEN,TIME, TIMESTAMP, TIMESTAMPDIFF, TIMESTAMPADD, TIMEZONE_HOUR, TIMEZONE_MINUTE, TO, TRAILING, TRIM,TRUE, UCASE, UNKNOWN, UPPER, VARBINARY, VARCHAR, VARYING, WHEN,WHENEVER, YEAR, ZONE.

Note – Even if an SQL term is on the list of allowed query terms, it is not guaranteed that the term can actually be used in a working query. The function of this list is to determine words that will not be treated as an identifier unless enclosed in double quotes.

SQL Words That Are Not Allowed in Queries

Some SQL reserved words (such as SELECT or CREATE) are forbidden from occurring unquoted in queries. The 5800 system server will immediately raise an exception when one of the forbidden reserved words is used in a query. These forbidden words can be used as attribute names by enclosing them in double quotes.

The following reserved words are forbidden:

ACTION, ADD, ALL, ALLOCATE, ALTER, ANY, APPLICATION, ARE, AREA, ASC, ASSERTION, ATOMIC, AUTHORIZATION, AVG, BEGIN, BY, CALL, CASCADE, CASCADED, CATALOG, CHECK, CLOSE, COLUMN, COMMIT, COMPRESS, CONNECT, CONNECTION, CONSTRAINT, CONSTRAINTS, CONTINUE, CONVERT, CORRESPONDING, CREATE, CROSS, CURRENT, CURRENT_PATH, CURRENT_SCHEMA, CURRENT_SCHEMAID, CURRENT_USER, CURRENT_USERID, CURSOR, DATA,

DEALLOCATE, DECLARE, DEFAULT, DEFERRABLE, DEFERRED, DELETE, DESC, DESCRIBE, DESCRIPTOR, DETERMINISTIC, DIAGNOSTICS, DIRECTORY, DISCONNECT, DISTINCT, DO, DOMAIN, DOUBLEATTRIBUTE, DROP, EACH, EXCEPT, EXCEPTION, EXEC, EXECUTE, EXTERNAL, FETCH, FLOAT, FOREIGN, FOUND, FULL, FUNCTION, GET, GLOBAL, GO, GOTO, GRANT, GROUP, HANDLER, HAVING, IDENTITY, IMMEDIATE, INDEX, INDEXED, INDICATOR, INITIALLY, INNER, INOUT, INPUT, INSENSITIVE, INSERT, INTERSECT, INTO, ISOLATION, JOIN, KEY, LANGUAGE, LAST, LEAVE, LEVEL, LOCAL, LONGATTRIBUTE, LOOP, MODIFIES, MODULE, NAMES, NATIONAL, NATURAL, NEXT, NULLIF, ON, ONLY, OPEN, OPTION, ORDER, OUT, OUTER, OUTPUT, OVERLAPS, OWNER, PARTIAL, PATH, PRECISION, PREPARE, PRESERVE, PRIMARY, PRIOR, PRIVILEGES, PROCEDURE, PUBLIC, READ, READS, REFERENCES, RELATIVE, REPEAT, RESIGNAL, RESTRICT, RETURN, RETURNS, REVOKE, ROLLBACK, ROUTINE, ROW, ROWS, SCHEMA, SCROLL, SECTION, SELECT, SEQ, SEQUENCE, SESSION, SESSION_USER, SESSION_USERID, SET, SIGNAL, SOME, SPACE, SPECIFIC, SQL, SQLCODE, SQLERROR, SQLEXCEPTION, SQLSTATE, SQLWARNING, STATEMENT, STRINGATTRIBUTE, SUM, SYSACC, SYSHGH, SYSLNK, SYSNIX, SYSTBLDEF, SYSTBLDSC, SYSTBT, SYSTBTATT, SYSTBTDEF, SYSUSR, SYSTEM_USER, SYSVIW, SYSVIWCOL, TABLE, TABLETYPE, TEMPORARY, TRANSACTION, TRANSLATE, TRANSLATION, TRIGGER, UNDO, UNION, UNIQUE, UNTIL, UPDATE, USAGE, USER, USING, VALUE, VALUES, VIEW, WHERE, WHILE, WITH, WORK, WRITE, ALLSCHEMAS, ALLTABLES, ALLVIEWS, ALLVIEWTEXTS, ALLCOLUMNS, ALLINDEXES, ALLINDEXCOLS, ALLUSERS, ALLTBTS, TABLEPRIVILEGES, TBTPRIVILEGES, MYSCHEMAS, MYTABLES, MYTBTS, MYVIEWS, SCHEMAVIEWS, DUAL, SCHEMAPRIVILEGES, SCHEMATABLES, STATISTICS, USRTBL, STRINGTABLE, LONGTABLE, DOUBLETABLE.

◆ ◆ ◆ CHAPTER 5

Programming Considerations and Best Practices

This chapter provides considerations and practices that can help you create efficient 5800 system applications.

The following topics are discussed:

Retries and Timeouts

Client applications should place API calls within retry loops to handle cases such as storage node failover. One immediate retry should be sufficient in the great majority of cases. In some cases of node failover, retries should be pursued for up to 30 seconds.

When the 5800 system server is sufficiently loaded, client timeouts may occur. To avoid this, maximum client threads should no greater than 25 times the number of nodes. For example, on a full-cell with 16 storage nodes, the maximum client threads should be less than or equal to 25 times 16, or 400 client threads.

Query Size Limit

There is a hard limit when querying a live 5800 system or hive on the combined size of query parameters and literal values that can be processed in one query. The limit is slightly over 8000 bytes. The query string itself does not count against this limit, just the parameter and literal values in the query.

Each additional parameter or literal in the query contributes 2 bytes of overhead plus the number of bytes to represent the query value. For this calculation, each date or time value consumes 4 bytes. Each long, double, or timestamp value consumes 8 bytes. A binary or char

field consumes the same number of bytes as the length of the value. A string value consumes twice as many bytes as the length of the value. These sizes are similar to what is described in the *Sun StorageTek 5800 System Administration Guide*, Table 7-6, Number of Bytes Used by Each Element Type in a Schema Table.

For example, assume dynamic parameter 1 is bound to string "Hello" in the query `system.test.type_string=?`. The parameter length would then be 12 bytes: 2 bytes overhead plus 10 bytes for a 5-character string parameter.

Conversely, note that in the query `system.test.type_string like '%Hello%'` that the parameter is 16 bytes: 2 bytes overhead plus 14 bytes for a 7 character string regardless of the declared size of the `system.test.type_string` field in the schema.

For the query `system.test.type_date = {date '2007-01-1'} AND system.test.type_char='Hello'`, the parameter length is 13 bytes: 2 bytes overhead for each of two query values plus 4 bytes for a date literal plus 5 bytes for a 5-character `char` parameter.

Limit the Size of Schema Query Parameters and Literals

When designing a schema, limit the sizes of fields so that planned queries will fit within the size limits. Most simple queries on a single table will fit within the 8000-byte limit. This is because the table row definition itself must fit within the similar 8080-bytes for overall table row size. Also note that complex queries on even a single table (such as queries involving many `OR` clauses) may overflow the limit. Schema design and query design must work together carefully if complex queries are an important part of the application.

For further information, see "Query Size Limit" on page 127.

Limit Results Per Fetch

Queries should use a reasonable value of "results per fetch" when a large total result set size is expected. The maximum result size should not exceed the memory allocated for the query engine on the server side. If it does, then the query will fail. Dropping the query result size will allow the query to succeed.

If you incorrectly estimate how much memory will be used by the result set, the server will not fail gracefully with an appropriate error message, but will instead run out of memory.

Suggested values are in the 2000-5000 range. Set `maxFetchSize = 4098` as an argument to the `query` method.

Index

N

O

P

Q

R

S

T