



Filters and Filter Strings

IdentityIQ Versions: 5.5, 6.0, 6.1, 6.2

This document describes specification of filter strings and Filter objects. Filters restrict the results of queries or constrain system actions to sets of objects that meet the filter criteria.

Document Revision History

| Revision Date | Written/Edited By | Comments |
|---------------|-------------------|---|
| July 2012 | Jennifer Mitchell | Initial Creation; current IdentityIQ version: 5.5p4 |
| August 2013 | Jennifer Mitchell | Updated for 6.1 compatibility (no changes to content for this); added Filter XML syntax info |
| Feb 2014 | Jennifer Mitchell | Updated for 6.2 compatibility (no content changes for this); added CompoundFilter syntax info |

© Copyright 2013 SailPoint Technologies, Inc., All Rights Reserved.

SailPoint Technologies, Inc. makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. SailPoint Technologies shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Restricted Rights Legend. All rights are reserved. No part of this document may be photocopied, reproduced, or translated to another language without the prior written consent of SailPoint Technologies. The information contained in this document is subject to change without notice.

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 for DOD agencies, and subparagraphs (c) (1) and (c) (2) of the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 for other agencies.

Regulatory/Export Compliance. The export and reexport of this software is controlled for export purposes by the U.S. Government. By accepting this software and/or documentation, licensee agrees to comply with all U.S. and foreign export laws and regulations as they relate to software and related documentation. Licensee will not export or reexport outside the United States software or documentation, whether directly or indirectly, to any Prohibited Party and will not cause, approve or otherwise intentionally facilitate others in so doing. A Prohibited Party includes: a party in a U.S. embargoed country or country the United States has named as a supporter of international terrorism; a party involved in proliferation; a party identified by the U.S. Government as a Denied Party; a party named on the U.S. Government's Entities List; a party prohibited from participation in export or reexport transactions by a U.S. Government General Order; a party listed by the U.S. Government's Office of Foreign Assets Control as ineligible to participate in transactions subject to U.S. jurisdiction; or any party that licensee knows or has reason to know has violated or plans to violate U.S. or foreign export laws or regulations. Licensee shall ensure that each of its software users complies with U.S. and foreign export laws and regulations as they relate to software and related documentation.

Trademark Notices. Copyright © 2013 SailPoint Technologies, Inc. All rights reserved. SailPoint, the SailPoint logo, SailPoint IdentityIQ, and SailPoint Identity Analyzer are trademarks of SailPoint Technologies, Inc. and may not be used without the prior express written permission of SailPoint Technologies, Inc. All other trademarks shown herein are owned by the respective companies or persons indicated.

Table of Contents

| | |
|--|----|
| Introduction..... | 5 |
| Filter Strings..... | 6 |
| Property Specification | 6 |
| Multi-valued Identity and Account Attributes | 6 |
| Datatype-Specific Syntax | 7 |
| Evaluation Operations | 7 |
| Basic Operators | 7 |
| Method Operators..... | 8 |
| Complex Query Method Operators..... | 8 |
| Grouping of Evaluations | 10 |
| Filter Objects | 11 |
| Operator Methods..... | 11 |
| Basic Operator Methods..... | 11 |
| Complex Operator Methods..... | 12 |
| Filter Modifiers | 13 |
| Filter Builder Helper Class | 13 |
| Adding Multiple Filter Conditions to a Filter Object | 15 |
| Adding a Single Criterion | 15 |
| Adding Filters with <i>And</i> | 15 |
| Adding Filters with <i>Or</i> | 16 |
| Adding Filters with Different Relationships..... | 16 |
| QueryOptions Objects | 18 |
| Adding a Filter to QueryOptions..... | 18 |
| Passing QueryOptions to a SailPointContext Method..... | 19 |
| Other Commonly Used QueryOptions/SailPointContext Features | 20 |
| Ordering with QueryOptions | 20 |
| Counting Objects with QueryOptions and SailPointContext | 21 |
| Filter Element in XML | 22 |
| Filter and Composite Filter Attributes..... | 22 |
| Examples of <Filter> XML | 23 |
| Creating XML Filters from FilterStrings and Filter Objects | 24 |

| | |
|---|----|
| CompoundFilter XML..... | 25 |
| Filtering by Identity Attributes | 25 |
| Filtering by Application Attributes | 26 |
| Filtering by Identity and Application Account Attributes..... | 28 |

Introduction

Filters are used in many places throughout IdentityIQ to allow actions to be applied to a subset of system objects. IdentityIQ contains a Filter object that includes a set of methods for structuring a filtering request. Many areas of IdentityIQ's user interface support specification of a "filter string" that can be turned into a Filter object by the system. The Filter object is then used to restrict the system action to only objects that match the criteria specified in the filter string.

Database queries are executed through the SailPointContext object. Options applicable to each query (including filters, ordering, grouping, etc.) are specified in a QueryOptions object. The filter option in a QueryOptions object is specified as a Filter object. To execute a query based on a Filter object, create a Filter object, add that Filter to a QueryOptions object, and specify that QueryOptions object in the SailPointContext method call that executes the query. When a filter string is entered in the UI, all of these steps are done automatically behind the scenes. When a query is run from a rule, all of these must be explicitly specified in the rule logic.

This document covers the specifics of writing filter strings, building Filter objects, and adding filters to a QueryOptions object. It includes example code to show how a query is executed from a rule through a SailPointContext object.

Filter Strings

In the IdentityIQ user interface, filters can be specified as filter strings that are then compiled by the filter compiler to create a Filter object.

Some examples of places where filter strings can be specified include:

- In the Attributes section of some Application Definitions (LDAP, ADAM) (Iterate Search Filter)
- On Lifecycle Events (Identity filter)
- On certain Tasks (Data Export, Identity Refresh, Policy Scan)
- On the Advanced Analytics window (Advanced Search option on Identity Search)

The filter compiler interprets a specific grammar whose syntax resembles Java. Its basic rules are:

- String literals must be in double quotes (e.g. department == **"Accounting"**)
- True/false (no quotes) are treated as Boolean literals (e.g. isActive == **true**)
- Numeric digits (no quotes) are treated as numbers
- Fully-qualified constants are resolved to enums (e.g. type == **CertificationItem.Type.Exception**)
- Everything else is assumed to be the property name

Property Specification

Only persistent properties in the object model can be specified in the query filter. In general, these are the properties available through the public "get" and "set" methods shown in the IdentityIQ Javadocs that ship with the product. The property names to specify match the method names without the "get"/"set" prefix. For example, the "first name" property is accessible through the getFirstname() method, so the property for the filter string would be **firstname** (the first letter of the property name is always lowercase; the rest matches the camel case of the method name). Attributes of objects contained within the Identity object can be queried with the object.attribute syntax (e.g. bundles.name or links.application.name).

Multi-valued Identity and Account Attributes

Most multi-valued attributes in the system can be queried directly through method operators (described in the *Method Operators* section below), but multi-valued Identity Attributes and Account Attributes are special cases that require a slightly more complex query syntax to access. The multi-valued Identity attributes can be accessed through the IdentityExternalAttribute object, and multi-valued Account attributes can be queried through the LinkExternalAttribute object using this syntax:

```
IdentityExternalAttribute.collectionCondition(  
    "((id.join(IdentityExternalAttribute.objectId) &&  
    IdentityExternalAttribute.attributeName i== \"IdentityAttributeName\" &&  
    IdentityExternalAttribute.value.startsWith(\"attributevalue\"))  
    )"  
)  
or
```

```
LinkExternalAttribute.collectionCondition(
    "( (links.id.join(LinkExternalAttribute.objectId) &&
    LinkExternalAttribute.attributeName i== \"AccountAttributeName\" &&
    LinkExternalAttribute.value.startsWith(\"attributevalue\")
    )) "
)
```

Example:

```
IdentityExternalAttribute.collectionCondition("
    ((id.join(IdentityExternalAttribute.objectId) &&
    IdentityExternalAttribute.attributeName i== \"costcenter\" &&
    IdentityExternalAttribute.value.startsWith(\"R01e\")
    )) "
)
```

The component statements in these queries (specifically, `collectionCondition` and `join`) are further described in the *Complex Query Method Operators* section of this document.

Datatype-Specific Syntax

The table below indicates the syntax required to add filters of various data types to the filter source.

| Field Data Types | Structure | Example |
|-------------------------|---|--|
| String | "value" | department == "Accounting" |
| Numeric | value | location <= 10 |
| Boolean | value | managerStatus == true |
| Date | DATE\$[long value of time – milliseconds since Jan 1, 1970] | lastLogin > DATE\$1318884600000 |
| Char (single character) | 'value' | middleInitial == 'D' |
| Float | Value (floating point literal) | average < 250.144 |
| Enumeration | EnumName.EnumValue | Type == CertificationItem.Type.Exception |

NOTE: The IdentityIQ object model currently has no persistent Char or Float fields, and it is rare for Enumerations to be queried through these pages. Those three data types are included here primarily as interesting information.

Evaluation Operations

Basic Operators

Object attributes can be evaluated against a specified value (or another field) using the basic equality/inequality operators, expressed like Java equality/inequality statements:

| Operation | Operator Symbol | Example |
|--------------|-----------------|--------------------------------|
| Equal | == | department == "Accounting" |
| Not equal | != | region != "Europe" |
| Less than | < | scorecard.compositeScore < 300 |
| Greater than | > | scorecard.compositeScore > 900 |

| | | |
|--------------------------|----|--------------|
| Less than or equal to | <= | tenure >= 10 |
| Greater than or equal to | >= | tenure <= 1 |

Any operator symbol can be prepended with “i” to request a case-insensitive comparison.

```
department i== "accounting"
region i!= "EUROPE"
```

Method Operators

These methods can be specified on the comparison attributes to invoke additional evaluation options:

| Method Operator | Description | Example |
|--|--|---|
| in() inIgnoreCase() | Property’s value is one of the specified string values | department.in({"Accounting", "Finance"}) |
| containsAll()* ** containsAllIgnoreCase()* ** | Property is a multi-valued field that contains all the values listed (field must be a list of normal datatypes, not objects) | certification.certifiers.containsAll({"Mark.Smith", "Susan.Jones"}) |
| isNull() | Property’s value is null | manager.isNull() |
| notNull() | Property’s value is not null | region.notNull() |
| isEmpty()* | Multi-valued property is empty | controlledScopes.isEmpty() |
| startsWith() startsWithIgnoreCase() | Property starts with the specified string value | location.startsWith("Lon") |
| endsWith() endsWithIgnoreCase() | Property ends with the specified string value | jobTitle.endsWithIgnoreCase("Clerk") |
| contains() containsIgnoreCase() | Property contains the specified string value somewhere within it | phone.contains("-454-") |

*These methods can be used with standard multi-valued attributes; multi-valued extended Identity Attributes and Account Attributes cannot use these methods and must instead use a collectionCondition query as illustrated in *Multi-valued Identity and Account Attributes* above.

** ContainsAll() and containsAllIgnoreCase() cannot currently be used on Identity filters in the UI. This is because all of the pre-configured multi-valued Identity attributes are objects (so the property values cannot be specified in string format for the comparison) and all custom configured multi-valued Identity attributes must be accessed through the IdentityExtendedAttributes object using the collectionCondition syntax.

NOTE: Queries using the contains() and endsWith() methods may be slow performing.

Complex Query Method Operators

Complex query filters can be specified using these additional method operators. They are specified on the comparison attributes just like the method operators discussed in the previous section.

| Method Operator | Description and Examples |
|---------------------|--|
| join (joinProperty) | Creates a connection between two objects based on the named properties NOTE: JoinProperty is case sensitive; the object name must be capitalized. Likewise, any additional criteria that apply to the second object must be |

| | |
|--|---|
| | <p>prefixed by the (case-sensitive) object name.</p> <p>Example:</p> <pre>department.join(Bundle.name) && Bundle.owner.name == "Walter.Henderson"</pre> <p>This is an identity search filter that joins to the Bundle object based on the Identity's department. It returns only Identities who are assigned to a department with a correspondingly named role that is owned by Walter Henderson.</p> |
| <p>collectionCondition (compound filter string)</p> | <p>Allows a separate filter string to be specified that will be executed against the specified property. This is useful when a multi-valued property is another object, such as Link, and multiple query criteria need to be imposed against attributes of that object to find the ones that should be used as filters against the primary object</p> <p>Example: Query for Identities with a privileged AD account</p> <pre>links.collectionCondition("application.name == \"AD\" && privileged == \"true\"")</pre> <p>This query cannot be specified with separate statements because <code>links.application.name == "AD" && links.privileged == true</code> would return any Identity who has any account on AD and any account that is privileged. For example, if Joe.Smith has a privileged account on RACF and a normal account on AD, he would still be in the return set from the normal "anded" statements but would not be returned from the collectionCondition query.</p> <p>NOTE: The entire condition must be expressed in quotes, and any quotes required within the condition must be escaped (<code>\</code>).</p> <p>The collectionCondition() method only applies – and only works – when the filter condition is a compound filter string (more than one condition is specified using <code>&&</code> or <code> </code>).</p> |
| <p>subquery (objectName, attribute name, subquery filter string)</p> | <p>Subquery runs a specified query on a secondary object, connected to the primary object by the property name and the secondary object's specified property.</p> <p>NOTE: ObjectName is case sensitive, and both the attribute name and subquery filter string must be specified in quotes. Quotes within the subquery filter string must be escaped (<code>\</code>).</p> <p>Example:</p> <pre>department.subquery(sailpoint.object.Bundle, "name", "owner.name i== \"Walter.Henderson\"")</pre> <p>This is another way of expressing the same filter that was described in the join method above.</p> |

Grouping of Evaluations

Evaluations can be grouped in sets using “and” or “or” operators. These are expressed in the filter string syntax with && and || respectively. Additionally, parentheses can be used to enforce a hierarchy to the grouping operations. Any condition can be negated by specifying ! before the parentheses.

| Example Statements | Evaluation Explanation |
|---|--|
| department == “Accounting” department.isNull() | Returns all Identities who are assigned to the Accounting department or who have no assigned department value (null department attribute) |
| (department == “Accounting” department.isNull()) && manager.name == “Adam.Smith” | Returns all Identities who report to manager Adam Smith and who are either in the Accounting department or have no assigned department |
| department == “Accounting” (department.isNull() && manager.name == “Adam.Smith”) | Returns all Identities who are <i>either</i> in the Accounting department <i>or</i> who report to Adam Smith and have no department value assigned |
| !(department == “Accounting”) | Returns all Identities who are in any department other than Accounting (i.e. not in Accounting) |

Filter Objects

A Filter object represents a set of query criteria, which can range from a simple, single condition to the most complex set of conditions that can be expressed in a SQL “where” clause. A Filter object is built by adding each of the required filter conditions to it. Each condition is specified through the appropriate operator method, and multiple filters are connected to each other using the “and” and “or” methods.

Operator Methods

Operator methods are used to specify each condition in the filter. They represent the basic evaluation options available on a SQL query.

Basic Operator Methods

The signature for the operators in this table is **[methodName](String propertyName, Object value)** (e.g. `eq(propertyName, value)`). The `like()` method can use this signature or can include specification of a third “matchMode” parameter, as shown.

| Operator Method | Description | Example |
|---|--|---|
| <code>eq ()</code> | Checks that the given property is equal to the given value | <code>Filter.eq("name", myName)</code> |
| <code>ne()</code> | Checks that the given property is not equal to the given value | <code>Filter.ne("owner", myIdentity)</code> |
| <code>lt(), le()</code> | Checks that the given property is less than / less than or equal to the given value | <code>Filter.lt("scorecard.compositeScore", maxRiskThreshold)</code> |
| <code>gt(), ge()</code> | Checks that the given property is greater than / greater than or equal to the given value | <code>Filter.ge("scorecard.compositeScore", riskTolerance)</code> |
| <code>like(propertyName, value)</code> or <code>like(propertyName, value, matchMode)</code> | Checks that the given property contains the value as a substring; location in the string is restricted by matchMode if provided. Possible matchModes are START, END, EXACT (equivalent to equals), and ANYWHERE (default if omitted) | <code>Filter.like("region", "America")</code> <code>Filter.like("application.name", "Enterprise", Filter.MatchMode.START)</code> |

The signature for the operators in the table below is **[methodName](String propertyName, Collection value)** (e.g. `in(propertyName, value)`).

| Operator Method | Description | Example* |
|----------------------------|---|---|
| <code>in()</code> | Checks that the given property has a value within the given set of values | <code>Filter.in("region", regions)</code> |
| <code>containsAll()</code> | Checks that the given multi-valued property contains all the given values | <code>Filter.containsAll("certifiers", certifiers)</code> |

*Regions and certifiers, in these examples, are Collections of strings.

The signature for the following operators is **[methodName](String propertyName)** (e.g. `notnull("region")`).

| Operator Method | Description | Example |
|------------------------|--|---|
| <code>notnull()</code> | Checks that the given property value is non-null | <code>Filter.notNull("manager")</code> |
| <code>isnull()</code> | Checks that the given property's value is null | <code>Filter.isnull("department")</code> |
| <code>isempty()</code> | Checks that the given multi-valued property is empty | <code>Filter.isempty("controlledScopes")</code> |

NOTE: Multi-valued Identity and Account (Link) Extended Attributes cannot be accessed through the `containsAll()` and `isEmpty()` methods because of the way these values are stored. A helper class – `ExternalAttributeFilterBuilder` – has been created to assist in building the required filter syntax for accessing these attributes. See *Filter Builder Helper Class* for more information.

Complex Operator Methods

These operator methods are used to specify more complex conditions in the filter.

- `join()` is used when a condition needs to be checked on a separate object and the only available reference to the second object is through an ID or other matching property, as opposed to a direct reference; it is usually specified in conjunction with another criterion that evaluates an attribute on the second object.
- `collectionCondition()` can be used to apply a compound filter to a second object that is available through a direct reference; for example, it can be used to specify match conditions for both the department and region for the selected identities' manager.
- `subquery()` functions similarly to `Join` in that it allows a condition to be checked on a separate object connected to the central object through a single value; the primary difference is that multiple subqueries can be performed on the same table while a join can only be done once per filter.

| Method Operator | Description and Examples |
|---|--|
| <code>join(String property, String joinProperty)</code> | Creates a connection between two objects based on the named properties; property is the name of the attribute on the primary object and joinProperty is the fully-qualified property name to join to on the second object. |
| | Example: <pre>Filter.join("department", "Bundle.name")</pre> |
| | This joins to the Bundle object based on its name matching the (Identity's) department. |
| <code>collectionCondition(String property, Filter compoundFilter)</code> | Allows a separate filter string to be specified that will be executed against the specified property. This is useful for a multi-valued property that is another object, such as Link, when multiple query criteria need to be imposed against attributes of that object to find the ones that should be used as filters against the primary object. |
| | Example: <pre>Filter adPrivileged = Filter.and(Filter.eq("application.name", "AD"), Filter.eq("privileged", "true")); Filter.collectionCondition("links", adPrivileged);</pre> |
| | This example returns Identities with a privileged AD account. Stating these criteria together in a collection condition prevents Identities with a non-privileged AD account but a privileged account on another system from being returned from the query. |

| | |
|--|--|
| | <p>NOTE: This method only applies – and only works – when the filter specified is a compound filter (involving an “and” or “or” condition).</p> <p>NOTE: If any Identity has more than one account on the application and one of them is privileged, this will return that Identity’s name twice. To get only one returned record per Identity, use the <code>setDistinct()</code> method on the <code>QueryOptions</code> object to which this filter is added (e.g. <code>qo.setDistinct(true)</code>).</p> |
| <code>subquery(String property, Class<?> subqueryClass, String subqueryProperty, Filter subqueryFilter)</code> | <p>Subquery runs a specified query on a secondary object, connected to the primary object by the property name and the secondary object’s specified property.</p> <p>Example:</p> <pre>Filter.subquery("department", Bundle.class, "name", Filter.ignoreCase(Filter.eq("owner.name", "Walter.Henderson")))</pre> <p>This joins to the Bundle object based on its name matching the (Identity’s) department; the subquery restricts the result set to only Bundles that are owned by Walter.Henderson, so the returned set of Identities from this filter would all be in departments with matching roles that are owned by Walter.Henderson.</p> |

Filter Modifiers

These two methods on the `Filter` object modify the `Filter` object specified as a parameter. Their signatures look like: **[methodName](Filter filter)** (e.g. `not(oldFilter)`).

| Operator Method | Description | Example |
|---------------------------|--|---|
| <code>not()</code> | Negates the specified filter | <code>Filter.not(oldFilter)</code> |
| <code>ignoreCase()</code> | Applies case-insensitivity to the specified filter | <code>Filter.ignoreCase(Filter.eq("department", "accounting"))</code> |

Filter Builder Helper Class

The filter object syntax for accessing multi-valued Identity and account (Link) extended attributes parallels the filter string syntax discussed in the filter string’s *Multi-valued Identity and Account Attributes* section. It uses the `collectionCondition` and `join` methods to connect from the Identity object to its `IdentityExternalAttribute` entries (or Link to `LinkExternalAttribute`).

It is possible to build this syntax manually to query these fields. However, SailPoint has provided a helper class for specifying these filters that makes it a little less cumbersome. That class is `ExternalAttributeFilterBuilder`.

NOTE: In a rule, you must import `sailpoint.search.ExternalAttributeFilterBuilder` to use this class’s methods.

All of these methods automatically create the required join from the primary table (Identity or Link) to the named table (**tableName**) based on a match between the connecting ID field on the primary object (**joinProperty**) and the “objectID” attribute on the named table. (The named table will either be `IdentityExternalAttribute` or `LinkExternalAttribute`.) Both the `buildAndFilter()` and `buildOrFilter()` methods provide two signatures: one with the final **op** parameter and one without it. If the **op** parameter is specified as “EQ”, it forces the comparison to `attrName` to be an *equals* evaluation instead of a *starts with* evaluation; if it is omitted or is anything other than “EQ”, *starts with* is used.

| ExternalAttributeFilterBuilder Method | Usage / Description |
|---|--|
| public static Filter buildAndFilter(String tableName, String joinProperty, String attrName, List<String> values, [String op]) | Looks for Identities (or Links) that have records containing <i>all</i> of the values in the list for the named attribute (attrName). Replicates a “containsAll” comparison. NOTE: The values list may contain as many or as few attribute values as should be in the containsAll list, so this can be used to look for one or more values. |
| public static Filter buildOrFilter(String tableName, String joinProperty, String attrName, List<String> values, [String op]) | Looks for Identities (or Links) that have records containing <i>any</i> of the values in the list for the named attribute (attrName). NOTE: Introducing multiple joins during an OR operation will have a negative impact on performance. |
| public static Filter buildNullFilters(String tableName, String joinProperty, String attrName) | Looks for Identities (or Links) with no value for the named attribute (attrName). |
| public static Filter buildNotNullFilters(String tableName, String joinProperty, String attrName) | Looks for Identities (or Links) with any non-null value in the named attribute (attrName). |

This example shows the syntax for manually building a Filter object that mimics a “containsAll(“R01e”, “L01e”)” condition for the multi-valued Identity extended attribute “costcenter”:

```
Filter f = Filter.and(Filter.join("id", "IdentityExternalAttribute.objectId"),
    Filter.eq("IdentityExternalAttribute.attributeName", "costcenter"));
f = Filter.and(f, Filter.eq("IdentityExternalAttribute.value", "R01e"));

Filter f2 = Filter.and(Filter.join("id", "IdentityExternalAttribute.objectId"),
    Filter.eq("IdentityExternalAttribute.attributeName", "costcenter"));
f2 = Filter.and(f2, Filter.eq("IdentityExternalAttribute.value", "L01e"));

f = Filter.and(f, f2);
Filter containsAll = Filter.collectionCondition("IdentityExternalAttribute", f);
```

This example uses the buildAndFilter() method to create the same containsAll(“R01e”, “L01e”) filter:

```
Filter containsAll = ExternalAttributeFilterBuilder.buildAndFilter("IdentityExternalAttribute", "id", "costcenter", Util.csvToList("R01e, L01e"), "EQ");
```

Constants are defined in the ExternalAttributeFilterBuilder class that can be used in these method calls if preferred.

| Constant | Value | Usage |
|-------------------|-----------------------------|--|
| IDENTITY_EXTERNAL | “IdentityExternalAttribute” | Specifies tableName for Identity join |
| IDENTITY_JOIN | “id” | Specifies joinProperty for Identity join |
| LINK_EXTERNAL | “LinkExternalAttribute” | Specifies tableName for Link join |
| LINK_JOIN | “links.id” | Speicfies joinProperty for Link join |

This example shows the same buildAndFilter() method call using the defined constants.

```
Filter containsAll = ExternalAttributeFilterBuilder.buildAndFilter(ExternalAttributeFilterBuilder.IDENTITY_EXTERNAL, ExternalAttributeFilterBuilder.IDENTITY_JOIN, "costcenter", Util.csvToList("R01e, L01e"), "EQ");
```

Adding Multiple Filter Conditions to a Filter Object

One or more filter conditions will be specified in any Filter object. Filter conditions can be added to the Filter object one at a time or several all at once, depending on the query criteria being specified. Some extremely complex combinations of “and” and “or” relationships are best added piecemeal for readability of the code being executed, but most criteria that relate to each other exclusively through “and” or exclusively through “or” can be added to the Filter all at one time.

Each method used to add criteria to a filter returns the Filter object including the newly added condition. Complex criteria can be built into a filter by adding more conditions to an existing Filter object. This is most easily understood by exploring it through examples.

Adding a Single Criterion

If only a single filter condition is being used, it is simply specified through its operator method. For example, consider a query that will retrieve an AccountGroup based on an application name, reference attribute, and native identity that are all known (and stored in variables myApp, myRefAttr, and myNativeIdent). Start with a basic filter that could return multiple account groups, based solely on the application:

```
Filter myFilter = Filter.eq("application.name", myApp);
```

Adding Filters with *And*

A second filter condition can be added to this existing filter using this syntax:

```
myFilter = Filter.and(myFilter, Filter.eq("referenceAttribute", myRefAttr));
```

This can be repeated as many times as necessary to append additional “and” conditions.

```
myFilter = Filter.and(myFilter, Filter.eq("referenceAttribute", myRefAttr));
myFilter = Filter.and(myFilter, Filter.ignoreCase(Filter.eq("nativeIdentity", myNativeIdent)));
```

Notice that an existing Filter object can have additional criteria added to it without having to create a new Filter object to hold the combined criteria. It is also possible to create a separate new Filter object as additional criteria are added; this can be useful if the original criterion is still needed as a stand-alone Filter.

```
Filter newFilter = Filter.and(myFilter, Filter.eq("referenceAttribute", myRefAttr));
```

The and() method offers another signature that can accept a List of Filter objects like this:

```
List filters = new ArrayList ();

filters.add(myFilter);
filters.add(Filter.eq("referenceAttribute", myRefAttr));
filters.add(Filter.ignoreCase(Filter.eq("nativeIdentity", myNativeIdent)));

myFilter = Filter.and(filters);
```

In custom Java code, multiple filters can be added to an “and” grouping at one time. This cannot be used in a rule since beanshell cannot process methods with variable arguments.

```
myFilter = Filter.and(myFilter,  
    Filter.eq("referenceAttribute", myRefAttr),  
    Filter.ignoreCase(Filter.eq("nativeIdentity", myNativeIdentity)));
```

However, beanshell can use the array syntax to add multiple conditions to the filter in a single statement like this:

```
myFilter = Filter.and(new Filter[] { myFilter,  
    Filter.eq("referenceAttribute", myRefAttr),  
    Filter.ignoreCase(Filter.eq("nativeIdentity", myNativeIdentity)) });
```

This builds the 3 filter conditions into an array and passes the array to the “and” method.

Adding Filters with *Or*

The “or” syntax mirrors the “and”, offering the same signatures: two Filter objects, a List of Filter objects, and (for custom Java code only, not beanshell) a variable number of Filter objects. Examples are shown below.

Two Filters:

```
Filter f = Filter.or(Filter.eq("bundles.id", roleId),  
    Filter.eq("assignedRoles.id", roleId));
```

List of Filters:

```
List appsList = new ArrayList();  
for (String app : includedAppIds) {  
    appsList.add(Filter.eq("links.application.id", app));  
}  
Filter f = (Filter.or(appsList));
```

Multiple Filters (Java only):

```
Filter f = Filter.or(Filter.ne("department", "Accounting"),  
    Filter.like("firstname", "A", Filter.MatchMode.START),  
    Filter.eq("region", myRegion));
```

Multiple Filters (beanshell):

```
Filter f1 = Filter.or(new Filter[] {  
    Filter.ne("department", "Accounting"),  
    Filter.like("firstname", "A", Filter.MatchMode.START),  
    Filter.eq("region", "Europe") });
```

Adding Filters with Different Relationships

Filter specification gets a little more complicated when some conditions are “or’d” together while others are “anded”. The way to do this is to specify the more closely tied filter conditions into one filter together first and then add the combination filter to the more loosely associated criteria.

For example, consider these query criteria for selecting an Identity (written in pseudocode):

(Last Name = “Smith”) OR (Department = Finance AND Manager = Catherine.Simmons)

The Department and Manager components jointly identify a single selection criterion so they must be grouped in a filter first. That filter can then be grouped with the Last Name component.

```
Filter f = Filter.and(Filter.eq("department", "Finance"),
```



```
        Filter.eq("manager.name", "Catherine.Simmons"));
f = Filter.or(f, Filter.eq("lastname", "Smith"));
```

Alternatively, this can be expressed in a single statement, as shown below.

```
Filter f = Filter.or(
    Filter.and(Filter.eq("department", "Finance"),
        Filter.eq("manager.name", "Catherine.Simmons")),
    Filter.eq("lastname", "Smith"));
```

Complex filter criteria may require more statements to build them into a Filter object. It is a good practice to minimize the complexity of each statement for readability and maintainability.

QueryOptions Objects

The QueryOptions object contains a variety of options that can affect the running of a query. Methods on the QueryOptions object allow specification of:

- Filter objects to restrict the results
- Attributes for ordering and grouping the returned results
- Options for ignoring case or requiring distinct results
- Limits on the number of records to return
- Location in an ordered list of the first record to return (used for paged results)
- Instructions on whether to apply configured scoping to the query and on how to extend scopes
- A pre-written HQL (Hibernate) query (supersedes any Filters specified)

The methods for these specifications are outlined in the IdentityIQ Javadocs. More information on the Scoping-related options is provided in the Scoping technical white paper on Compass. The Filter-specific methods are further discussed below.

Adding a Filter to QueryOptions

Filters can be added to QueryOptions through its addFilter() method.

```
QueryOptions ops = new QueryOptions();
ops.addFilter(Filter.eq("application.name", myApp));
```

The filter can be built as it is added to the QueryOptions object, as shown above, or it can be built first and added after it is complete.

```
Filter myFilter = Filter.and(Filter.eq("application.name", myApp),
                             Filter.eq("referenceAttribute", myRefAttr));
QueryOptions ops = new QueryOptions();
ops.addFilter(myFilter);
```

When multiple separate filters are added to one QueryOptions object, they are “anded” together when the query runs.

```
QueryOptions qo = new QueryOptions();
qo.addFilter(Filter.eq("application.name", myApp));
qo.addFilter(Filter.eq("referenceAttribute", myRefAttr));
```

Custom java code can use the QueryOptions constructor or its add() method to add one or more filters. Rules cannot use these because beanshell cannot process variable arguments.

```
QueryOptions ops = new QueryOptions(Filter.eq("application.name", myApp));

or

QueryOptions qo = new QueryOptions();
qo.add(Filter.eq("application.name", myApp), Filter.eq("referenceAttribute",
myRefAttr));
```

Multiple filters can be added at one time to a QueryOptions object in beanshell by building a list of filters and using the setFilters() method. This replaces any existing filters in the QueryOptions object.

```
List filters = new ArrayList();
filters.add(Filter.eq("application.name", myApp));
filters.add(Filter.eq("referenceAttribute", myRefAttr));

QueryOptions qo = new QueryOptions();
qo.setFilters(filters);
```

Passing QueryOptions to a SailPointContext Method

Once all the desired options are included in the QueryOptions object, the query can be executed. All IdentityIQ rules include a parameter called **context** that is a SailPointContext object. SailPointContext offers two different methods for executing queries, both of which take a QueryOptions object as a parameter.

| SailPointContext Method | Description |
|---|--|
| public Iterator search (Class cls, QueryOptions options, List<String> properties) | Returns an iterator of the requested properties from the objects matching the query e.g. context.search(Identity.class, qo, "id, name") returns an iterator of the ids and names from the Identity objects matching the query terms in the qo QueryOptions object |
| public List getObjects (Class cls, QueryOptions options) | Returns a list of objects (of the specified class) matching the query criteria |

The example below queries for an accountGroup object based on the specified filter and prints the names of all the returned account groups.

```
import sailpoint.object.*;

String myApp = "ADAM";
String myRefAttr = "groups";

Filter myFilter = Filter.and(Filter.eq("application.name", myApp),
                             Filter.eq("referenceAttribute", myRefAttr));

QueryOptions qo = new QueryOptions();
qo.addFilter(myFilter);

List myAcctGrps = context.getObjects(AccountGroup.class, qo);

if (null != myAcctGrps) {
    for (AccountGroup acctGrp : myAcctGrps) {
        System.out.println("Account Group Name = "+ acctGrp.getName());

        // You have to be careful with other code, but this can be
        // important for performance.
        context.decach();
    }
}
```

NOTE: It is a best practice, for performance reasons, to decache occasionally when looping over a large number of objects.

This next example uses the context's search method to return an iterator that contains the names of the account groups, rather than a list of full objects, and then prints those names.

```
import sailpoint.object.*;

String myApp = "ADAM";
String myRefAttr = "groups";

Filter myFilter = Filter.and(Filter.eq("application.name", myApp),
                             Filter.eq("referenceAttribute", myRefAttr));

QueryOptions qo = new QueryOptions();
qo.addFilter(myFilter);

List props = new ArrayList();
props.add("name");

Iterator names = context.search(AccountGroup.class, qo, props);
if (null != names) {
    while (names.hasNext()) {
        String name = (String) names.next()[0];

        System.out.println("Account Group name = " + name);
    }
}
```

NOTE: It is a best practice to **avoid** breaking out of loops when iterating over search results. Doing so can leave open database cursors. Two examples of what **not** to do are shown below.

```
// Don't do this!
Boolean quitLooping = false;
if (null != names && !quitLooping) {
    while (names.hasNext()) {
        String name = (String) names.next()[0];

        System.out.println("Account Group name = " + name);

        // Do not break out of a loop before getting
        // to the end of the search results like this.
        if ("Special Group".equals(name)) {
            break;
        }

        // Do not break out using a loop condition like this either.
        if ("Special Group".equals(name)) {
            quitLooping = true;
        }
    }
}
```

Other Commonly Used QueryOptions/SailPointContext Features

Ordering with QueryOptions

After filters, ordering is one of the most frequently used features of QueryOptions. This is an example of the Java code that applies ordering to a search. This example orders results by lastname, then firstname, then ID.

```
List<QueryOptions.Ordering> ordering = new ArrayList<QueryOptions.Ordering>();
ordering.add(new QueryOptions.Ordering("lastname", true));
ordering.add(new QueryOptions.Ordering("firstname", true));
ordering.add(new QueryOptions.Ordering("id", true));

QueryOptions queryOps = new QueryOptions();
queryOps.setOrderings(ordering);
```

Like filters, ordering can only be based on persistent object attributes.

Counting Objects with QueryOptions and SailPointContext

It can often be helpful to get a count of objects returned by a set of query criteria. A built-in method in the `SailPointContext` – `countObjects` – makes that easy to do. `CountObjects`' signature is:

```
public int countObjects (Class cls, QueryOptions options)
```

The code below illustrates how to pass a `queryOptions` object to this method; the method returns an integer count of the objects meeting the criteria in that `queryOptions`.

```
QueryOptions qo = new QueryOptions();
// ... add filters to the query options as described above ...

int total = ctx.countObjects(CertificationItem.class, qo);

// total now contains the number of records returned by the query.
```

Filter Element in XML

Filters can also be represented in XML in IdentityIQ. The primary usage for this is to specify the filters that define a GroupDefinition object. A GroupDefinition represents either a Group generated from a GroupFactory or a Population created from a query. The GroupDefinition contains a GroupFilter object which houses the group or population's membership criteria in the form of a filter.

A single condition is represented as a <Filter> element. Multiple Filter elements can be combined together through a <CompositeFilter> element.

```
<GroupFilter>
  <Filter operation="EQ" property="Department" value="Accounting"/>
</GroupFilter>

<GroupFilter>
  <CompositeFilter operation="OR">
    <Filter operation="EQ" property="Department" value="Accounting"/>
    <Filter operation="EQ" property="Region" value="Europe"/>
  </CompositeFilter>
</GroupFilter>
```

Filter and Composite Filter Attributes

Attributes on the Filter element are defined in the table below.

| Attribute | Usage |
|------------------|---|
| operation | Specifies the operation (equivalent of <i>Operator Methods</i> in Filter objects) Valid values: EQ, NE, LT, GT, LE, GE, IN, CONTAINS_ALL, LIKE, NOTNULL, ISNULL, ISEMPTY, JOIN, COLLECTION_CONDITION |
| property | Specifies the attribute being evaluated |
| value | Specifies the value to look for or compare to in the property; used with the comparative operations (EQ, NE, LT, GT, LE, GE, IN, CONTAINS_ALL, LIKE) |
| ignoreCase | Boolean value indicating whether the comparison should be case-sensitive or not |
| joinProperty | Used with JOIN operation; specifies the property on another objects that should be used to join to the property on the primary object e.g. <Filter joinProperty="IdentityExternalAttribute.objectId" operation="JOIN" property="id"/> performs a join between identity and identityExternalAttribute based on identity.id = identityExternalAttribute.objectId |
| matchMode | Used with the LIKE operation to specify where to look for the match; Valid values: START, END, ANYWHERE, EXACT |
| subqueryClass | Contains the fully-qualified class name of the the class to which a specified subquery applies (only specified with SubqueryFilter (see <i>Nested Element</i>) |
| subqueryProperty | Names the property returned from the subquery class, which is then compared to primary filter Property according to the primary query operator |

Additionally, these nested elements can be specified to help define the Filter.

| Nested Element | Usage |
|---------------------|--|
| CollectionCondition | Specified when the operation is COLLECTION_CONDITION; specified as a CompositeFilter object |
| SubqueryFilter | Specifies the filter for a subquery; this can be specified either as a Filter object or a CompositeFilter object (depending on its complexity) |
| Value | Alternative to the value attribute on the Filter element; specifies the value to use in the operation; used with comparative operations |

CompositeFilters elements are only used to join other filters or to modify a filter with a NOT operation. They contain only a single attribute: operation, and its value can be set to NOT, AND, or OR. They can contain as many nested Filters or CompositeFilters as are necessary to specify the filter criteria.

| Attribute | Usage |
|-----------|--|
| operation | Specifies the operation for combining (or modifying) the Filters and CompositeFilters contained within it. Valid values are: NOT, AND, OR. |

Examples of <Filter> XML

This section shows the XML representations of filters; most of these are based on examples shown as Filter object or filter string representations in other sections of this document.

Simple Equality Filter: Returns all identities with attribute Department = "Accounting"

```
<Filter operation="EQ" property="department" value="Accounting"/>
```

Collection Condition Filter: Returns all identities with a privileged account on the Active Directory application (extended3 is the link extended attribute where the "privileged" attribute value is stored)

```
<Filter operation="COLLECTION_CONDITION" property="links">
  <CollectionCondition>
    <CompositeFilter operation="AND">
      <Filter operation="EQ" property="application.name" value="Active_Directory"/>
      <Filter operation="EQ" property="extended3" value="true"/>
    </CompositeFilter>
  </CollectionCondition>
</Filter>
```

Subquery Filter: Joins to Bundle object based on the Identity's department and returns only Identities who are assigned to a department with a correspondingly named role that is owned by Walter Henderson

```
<Filter operation="IN" property="department" subqueryClass="sailpoint.object.Bundle"
subqueryProperty="name">
  <SubqueryFilter>
    <Filter operation="EQ" property="owner.name" value="Walter.Henderson"/>
  </SubqueryFilter>
</Filter>
```

Comparative Operator Filter: Returns all Identities in the Accounting or Finance departments (illustrates the nested <Value> element's usage)

```
<Filter operation="IN" property="Department">
  <Value>
    <List>
      <String>Accounting</String>
      <String>Finance</String>
    </List>
  </Value>
</Filter>
```

Composite Filter: Returns all active Identities who are either in Austin or are in the Marketing department

```
<CompositeFilter operation="AND">
  <Filter operation="EQ" property="inactive" value="false"/>
  <CompositeFilter operator="OR">
    <Filter operation="EQ" property="department" value="Marketing"/>
    <Filter operation="EQ" property="location" value="Austin"/>
  </CompositeFilter>
</CompositeFilter>
```

Creating XML Filters from FilterStrings and Filter Objects

XML representations of Identity-related filter are easily generated from FilterStrings through the Advanced Analytics Identity Search page. Any query specified on the Identity Search page (whether specified as a FilterString or through UI-selected criteria) can be saved as a population. This creates a GroupDefinition object which contains an XML representation of the filter criteria.

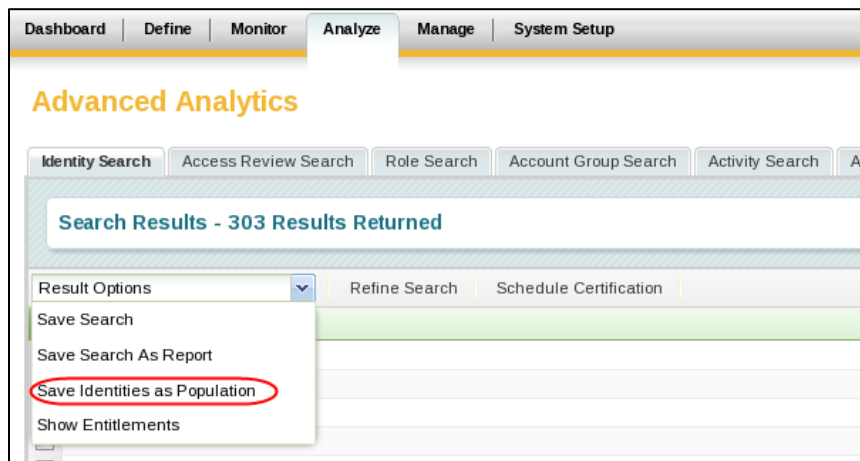


Figure 1: Save Filter in GroupDefinition XML by Saving Identities as Population

XML filter representations can also be generated from a Filter object from any rule, which can be a helpful debugging tool. The Filter object contains a `toXml()` method that generates the filter as an XML object. This can be written to `stdout` with a `System.out.println()` statement, saved to a Custom object, etc. The example rule below generates the Filter XML for the subquery filter object example shown in the Examples section above.

```
import sailpoint.object.*;

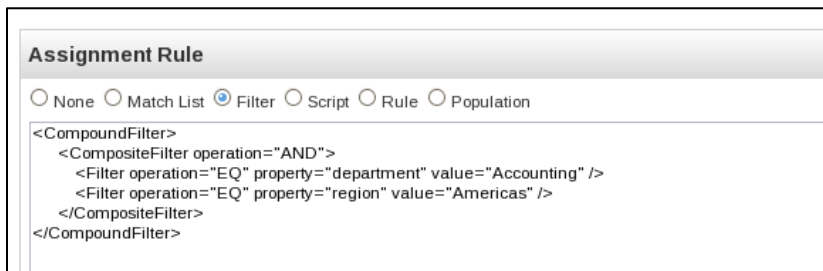
Filter myFilter = Filter.subquery("department", Bundle.class, "name",
Filter.ignoreCase(Filter.eq("owner.name", "Walter.Henderson")));

System.out.println("Filter: " + myFilter.toXml());
```


Additionally, when the target of the serialized filter is the Identity class, the resulting XML can be used to create GroupDefinition objects manually, if desired.

CompoundFilter XML

A CompoundFilter is used for specifying identity selectors in several places in IdentityIQ; it is offered as the “Filter” option, alongside the choices: Match List, Script, Rule, or Population. For example, this collection of options is offered for restricting Lifecycle Events or Policies to subsets of Identities and for specifying Role Assignment Rules.



Assignment Rule

☐ None
 ☐ Match List
 ☒ Filter
 ☐ Script
 ☐ Rule
 ☐ Population

```

<CompoundFilter>
  <CompositeFilter operation="AND">
    <Filter operation="EQ" property="department" value="Accounting" />
    <Filter operation="EQ" property="region" value="Americas" />
  </CompositeFilter>
</CompoundFilter>
  
```

Figure 2: Example Filter Identity Selector Option (as a Role Assignment Rule)

To use the Filter option in these cases, the administrator must specify a CompoundFilter in its XML format. CompoundFilters can reference identity attributes, application account attributes, or both. They are made up of a Filter (which can be expressed as a Composite Filter to combine multiple Filters together) and an optional Applications list. The applications list is only needed when the filter references application account attribute; identity attributes can be examined without needing that applications list.

NOTE: Though this CompoundFilter construct is available and can be useful for specifying identity selection criteria in a Filter representation, it is the least commonly used choice. Match Expressions are generally the most popular because of their relative simplicity, Scripts and Rules are commonly used for more complex criteria, and Populations are more easily used to apply Filter-based criteria to the action.

Filtering by Identity Attributes

When the filter references only Identity attributes, a CompoundFilter is really just a wrapper for a Filter object. (This Filter is often a CompositeFilter that groups other Filters together). These two examples illustrate how to reference identity attributes in a CompoundFilter; the first is fairly simple and the second is more complex to illustrate how different condition relationships can be expressed.

Example 1: Apply the action to any identity in the Accounting Department and the Americas Region.

```

<CompoundFilter>
  <CompositeFilter operation="AND">
    <Filter operation="EQ" property="department" value="Accounting" />
    <Filter operation="EQ" property="region" value="Americas" />
  </CompositeFilter>
</CompoundFilter>
  
```

Example 2: Apply the action to any active identity who is in business unit 123, 234, or 456 *or* to any active identity with job code 100, 040, or 046.

```
<CompoundFilter>
  <CompositeFilter operation="OR">
    <CompositeFilter operation="AND">
      <Filter operation="NE" property="inactive">
        <Value>
          <Boolean>true</Boolean>
        </Value>
      </Filter>
      <Filter operation="IN" property="BusinessUnit">
        <Value>
          <List>
            <String>123</String>
            <String>234</String>
            <String>456</String>
          </List>
        </Value>
      </Filter>
    </CompositeFilter>
    <CompositeFilter operation="AND">
      <Filter operation="NE" property="inactive">
        <Value>
          <Boolean>true</Boolean>
        </Value>
      </Filter>
      <Filter operation="IN" property="job_code">
        <Value>
          <List>
            <String>100</String>
            <String>040</String>
            <String>046</String>
          </List>
        </Value>
      </Filter>
    </CompositeFilter>
  </CompositeFilter>
</CompoundFilter>
```

Filtering by Application Attributes

To reference application account attributes in the CompoundFilter, the application must be referenced in the property attribute of the Filter. This can be done in one of two ways: by specifying the application name in the property or by including an Applications list in the CompoundFilter and referencing the application by number.

The Applications list and numeric references are more commonly used because they are often better for long-term maintainability. The numeric references are indexes to the application's position in the CompoundFilter's Applications list. **NOTE:** The index position numbers start with 0, not 1.

Example: Apply the action to identities who have an Active Directory account that includes a groupmbr value "InvntryAnalysis" and who have either an ERP_Global account or an ERP_Global_Platform account with a groupmbr value of "PayrollAnalysis".

```
<CompoundFilter>
  <Applications>
```

```

    <Reference class="sailpoint.object.Application" name="Active_Directory"/>
    <Reference class="sailpoint.object.Application" name="ERP_Global"/>
    <Reference class="sailpoint.object.Application"
name="Composite_ERP_Global_Platform"/>
  </Applications>
  <CompositeFilter operation="AND">
    <Filter matchMode="ANYWHERE" operation="CONTAINS_ALL" property="0:groupmbr">
      <Value>
        <List>
          <String>InvntryAnalysis</String>
        </List>
      </Value>
    </Filter>
    <CompositeFilter operation="OR">
      <Filter matchMode="ANYWHERE" operation="CONTAINS_ALL" property="1:groupmbr">
        <Value>
          <List>
            <String>PayrollAnalysis</String>
          </List>
        </Value>
      </Filter>
      <Filter matchMode="ANYWHERE" operation="CONTAINS_ALL" property="2:groupmbr">
        <Value>
          <List>
            <String>PayrollAnalysis</String>
          </List>
        </Value>
      </Filter>
    </CompositeFilter>
  </CompositeFilter>
</CompoundFilter>

```

The Applications list can be omitted and the Application names can be directly specified in the property attribute of the Filter only if there are no spaces in the application names.

Example: Apply the action to any identity with an Active Directory account with a groupmbr attribute containing PayrollAnalysis.

```

<CompoundFilter>
  <Filter matchMode="ANYWHERE" operation="CONTAINS_ALL"
property="Active_Directory:groupmbr">
    <Value>
      <List>
        <String>PayrollAnalysis</String>
      </List>
    </Value>
  </Filter>
</CompoundFilter>

```

The primary reason the Applications list syntax is recommended over this direct name syntax is that, if an application name changes, Filters whose property references the application by name must be manually updated. By contrast, IdentityIQ uses the application ID to access applications referenced in the Applications list, so CompoundFilters using the Applications list still work even if a referenced application is renamed.

NOTE: The two examples above specified the Value for the Filter as a List. Filter XML generated by IdentityIQ is consistently created using this List format, but this is not strictly necessary in these cases since there is only one

value in the list. If it were manually created by an administrator, example 2 could be more concisely written like this:

```
<CompoundFilter>
  <Filter matchMode="ANYWHERE" operation="CONTAINS_ALL"
property="Active_Directory:groupmbr" value='PayrollAnalysis' />
</CompoundFilter>
```

Filtering by Identity and Application Account Attributes

CompoundFilters can contain references to both Identity attributes and application account attributes at the same time. Any property that does not contain an application reference (either by name or number) is interpreted as referencing an Identity attribute.

Example: Apply the action to any Identity in the Accounting Department whose Active Directory account contains groupmbr value “PayrollAnalysis”.

```
<CompoundFilter>
  <Applications>
    <Reference class="sailpoint.object.Application" name="Active_Directory" />
  </Applications>
  <CompositeFilter operation="AND">
    <Filter operation="EQ" property="department" value="Accounting" />
    <Filter matchMode="ANYWHERE" operation="CONTAINS_ALL" property="0:groupmbr"
value='PayrollAnalysis' />
  </CompositeFilter>
</CompoundFilter>
```