

SOQL and SOSL Reference

Version 46.0, Summer '19





CONTENTS

Chapter 1: Introduction to SOQL and SOSL
Chapter 2: Salesforce Object Query Language (SOQL)
Typographical Conventions in This Document
Quoted String Escape Sequences
Reserved Characters
Alias Notation
SOQL SELECT Syntax
Condition Expression Syntax (WHERE Clause)
fieldExpression Syntax
USING SCOPE
ORDER BY
LIMIT
OFFSET
Update an Article's Keyword Tracking with SOQL
Update an Article Viewstat with SOQL
WITH filteringExpression
GROUP BY
HAVING
TYPEOF41
FORMAT ()
FOR VIEW
FOR REFERENCE
FOR UPDATE
Aggregate Functions
Date Functions
Querying Currency Fields in Multi-Currency Orgs
Example SELECT Clauses
Relationship Queries
Understanding Relationship Names
Using Relationship Queries
Understanding Relationship Names, Custom Objects, and Custom Fields
Understanding Query Results
Lookup Relationships and Outer Joins
Identifying Parent and Child Relationships
Understanding Relationship Fields and Polymorphic Fields
Understanding Relationship Query Limitations
Using Relationship Queries with History Objects
Using Relationship Queries with Data Category Selection Objects

Contents

Using Relationship Queries with the Partner WSDL
Change the Batch Size in Queries
SOQL Limits on Objects
SOQL with Big Objects
Syndication Feed SOQL and Mapping Syntax
Location-Based SOQL Queries
Async SOQL
Chapter 3: Salesforce Object Search Language (SOSL)
Typographical Conventions in This Document
SOSL Limits
SOSL Limits on External Objects
SOSL Syntax
Example Text Searches
convertCurrency()
FIND {SearchQuery}
FORMAT()
IN SearchGroup
LIMIT n
OFFSET n
ORDER BY Clause
RETURNING FieldSpec
toLabel(fields)
Update an Article's Keyword Tracking with SOSL
Update an Article's Viewstat with SOSL
USING Listview=
WHERE conditionExpression
WITH DATA CATEGORY DataCategorySpec
WITH DivisionFilter
WITH HIGHLIGHT
WITH METADATA
WITH NETWORK NetworkldSpec
WITH Pricebookld
WITH SNIPPET
WITH SPELL_CORRECTION
Indox

CHAPTER 1 Introduction to SOQL and SOSL

If you've built a custom UI for Salesforce, you can use the Salesforce Object Query Language (SOQL) and Salesforce Object Search Language (SOSL) APIs to search your organization's Salesforce data.

This guide explains when to use SOQL and SOSL and outlines the syntax, clauses, limits, and performance considerations for both languages. It is intended for developers and assumes knowledge and experience working with APIs to interact with data.

Deciding Which to Use

A SOQL query is the equivalent of a SELECT SQL statement and searches the org database. SOSL is a programmatic way of performing a text-based search against the search index.

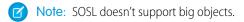
Whether you use SOQL or SOSL depends on whether you know which objects or fields you want to search, plus other considerations.

Use SOQL when you know which objects the data resides in, and you want to:

- Retrieve data from a single object or from multiple objects that are related to one another.
- Count the number of records that meet specified criteria.
- Sort results as part of the query.
- Retrieve data from number, date, or checkbox fields.

Use SOSL when you don't know which object or field the data resides in, and you want to:

- Retrieve data for a specific term that you know exists within a field. Because SOSL can tokenize
 multiple terms within a field and build a search index from this, SOSL searches are faster and can
 return more relevant results.
- Retrieve multiple objects and fields efficiently where the objects might or might not be related to one another.
- Retrieve data for a particular division in an organization using the divisions feature.
- Retrieve data that's in Chinese, Japanese, Korean, or Thai. Morphological tokenization for CJKT terms helps ensure accurate results.



Performance Considerations

To increase the efficiency of queries and searches, keep in mind:

- Both SOQL WHERE filters and SOSL search queries can specify text you should look for. When a
 given search can use either language, SOSL is generally faster than SOQL if the search expression
 uses a CONTAINS term.
- SOSL can tokenize multiple terms within a field (for example, multiple words separated by spaces)
 and builds a search index off this. If you're searching for a specific distinct term that you know exists
 within a field, you might find SOSL is faster than SOQL for these searches. For example, you might

use SOSL if you were searching for "John" against fields that contained values like "Paul and John Company".

• Keep the number of fields to be searched or queried to a minimum. Using a large number of fields leads to a large number of permutations, which can be difficult to tune.

For more information, see Best Practices for Deployments with Large Data Volumes.

Sending Queries

Use the REST and SOAP protocols to execute queries and searches:

- Query (REST) and query () (SOAP)—Executes a SOQL query against the specified object and returns data that matches the specified criteria.
- Search (REST) and search () (SOAP)—Executes a SOSL text string search against your org's
 data

More resources to perform other common search tasks, like auto-suggesting records, articles, and queries, are also available.

In Apex, you can use SOQL or SOSL on the fly by surrounding the statement in square brackets. You can also use a Search Class to perform dynamic SOSL queries and a Search Namespace for getting search results and suggestion results.



Note: Apex requires that you surround SOQL and SOSL statements with square brackets to use them on the fly. You can use Apex script variables and expressions when preceded by a colon (:).

CHAPTER 2 Salesforce Object Query Language (SOQL)

In this chapter ...

- Typographical Conventions in This Document
- Quoted String Escape Sequences
- Reserved Characters
- Alias Notation
- SOQL SELECT Syntax
- Relationship Queries
- Change the Batch Size in Queries
- SOQL Limits on Objects
- Syndication Feed SOQL and Mapping Syntax
- Location-Based SOQL Queries
- Async SOQL

Use the Salesforce Object Query Language (SOQL) to search your organization's Salesforce data for specific information. SOQL is similar to the SELECT statement in the widely used Structured Query Language (SQL) but is designed specifically for Salesforce data.

With SOQL, you can construct simple but powerful guery strings in the following environments:

- In the queryString parameter in the query () call
- In Apex statements
- In Visualforce controllers and getter methods
- In the Schema Explorer of the Force.com IDE

Similar to the SELECT command in Structured Query Language (SQL), SOQL allows you to specify the source object (such as Account), a list of fields to retrieve, and conditions for selecting rows in the source object.



Note: SOQL doesn't support all advanced features of the SQL SELECT command. For example, you can't use SOQL to perform arbitrary join operations, use wildcards in field lists, or use calculation expressions.

SOQL uses the SELECT statement combined with filtering statements to return sets of data, which can optionally be ordered:

SELECT one or more fields

FROM an object

WHERE filter statements and, optionally, results are ordered

For example, the following SOQL query returns the value of the Id and Name field for all Account records if the value of Name is Sandy:

SELECT Id, Name FROM Account WHERE Name = 'Sandy'



Note: Apex requires that you surround SOQL and SOSL statements with square brackets to use them on the fly. You can use Apex script variables and expressions when preceded by a colon (:).

For a complete description of the syntax, see SOQL SELECT Syntax.

When to Use SOQL

Use SOQL when you know which objects the data resides in, and you want to:

- Retrieve data from a single object or from multiple objects that are related to one another.
- Count the number of records that meet specified criteria.
- Sort results as part of the query.
- Retrieve data from number, date, or checkbox fields.

Salesforce Object Query Language (SOQL)



Note: With archived data and big objects, you can use only some SOQL features. For more information, see SOQL with Big Objects on page 74.

Typographical Conventions in This Document

This SOQL reference uses custom typographical conventions.

Convention	Description
SELECT Name FROM Account	Courier font indicates items that you should type as shown. In a syntax statement, Courier font also indicates items that you should type as shown, except for curly braces, square brackets, ellipsis, and other typographical markers explained in this table.
SELECT fieldname FROM objectname	Italics represent a variable or placeholder. You supply the actual value.
{}	Curly braces group elements to remove ambiguity. For example, in the clause UPDATE {TRACKING VIEWSTAT}[,], the curly braces indicate that the pipe shows a choice between TRACKING and VIEWSTAT after UPDATE, rather than a choice between UPDATE TRACKING and VIEWSTAT.
I	The pipe character separates alternate elements. For example, in the clause UPDATE {TRACKING VIEWSTAT}[,], the character indicates that you can use either TRACKING or VIEWSTAT after UPDATE.
[]	Square brackets indicate an optional element. For example, [LIMIT rows] means that you can specify zero or one LIMIT clause. Don't type square brackets as part of a SOQL command. Nested square brackets indicate elements that are optional and can only be used if the parent optional element is present. For example, in the clause [ORDER BY fieldOrderByList [ASC DESC] [NULLS {FIRST LAST}]], ASC, DESC, or the NULLS clause cannot be used without the ORDER BY clause.
[] and [,]	Square brackets containing an ellipsis indicate that the preceding element can be repeated up to the limit for the element. If a comma is also present, the repeated elements must be separated by commas. If the element is a list of choices grouped with curly braces, you can use items from the list in any order. For example, in the clause UPDATE {TRACKING VIEWSTAT}[,], the [,] indicates that you can use TRACKING, VIEWSTAT, or both:
	UPDATE TRACKING
	UPDATE VIEWSTAT
	UPDATE TRACKING, VIEWSTAT

Quoted String Escape Sequences

SOQL defines several escape sequences that are valid in queries so that you can include special characters in your queries. You can escape new lines, carriage returns, tabs, quotes, and more. The escape character for SOQL is the backslash (\) character.

You can use the following escape sequences with SOQL:

Sequence	Meaning
\n or \N	New line
\r or \R	Carriage return
\t or \T	Tab
\b or \B	Bell
\f or \F	Form feed
\"	One double-quote character
\'	One single-quote character
\\	Backslash
LIKE expression only: _	Matches a single underscore character (_)
LIKE expression only:\%	Matches a single percent sign character (%)

If you use a backslash character in any other context, an error occurs.

Escaped Character Examples

SELECT Id FROM Account WHERE Name LIKE 'Ter%'

Select all accounts whose name begins with the three character sequence 'Ter'.

SELECT Id FROM Account WHERE Name LIKE 'Ter\%'

Select all accounts whose name exactly matches the four character sequence 'Ter%'.

SELECT Id FROM Account WHERE Name LIKE 'Ter\%%'

Select all accounts whose name begins with the four character sequence 'Ter%'

Reserved Characters

The single quote (') and backlash (\) characters are reserved in SOQL queries and must be preceded by a backslash to be properly interpreted.

Reserved characters, if specified in a SELECT clause as a literal string (between single quotes), must be escaped (preceded by the backslash \ character) to be properly interpreted. An error occurs if you do not precede reserved characters with a backslash.

The following characters are reserved:

' (single quote)
\ (backslash)

For example, to query the Account Name field for "Bob's BBQ," use the following SELECT statement:

SELECT Id
FROM Account
WHERE Name LIKE 'Bob\'s BBQ'

Alias Notation

You can use alias notation in SELECT queries.

```
SELECT count()

FROM Contact c, c.Account a

WHERE a.name = 'MyriadPubs'
```

To establish the alias, first identify the object, in this example a contact, and then specify the alias, in this case "c." For the rest of the SELECT statement, you can use the alias in place of the object or field name.

The following are SOQL keywords that can't be used as alias names: AND, ASC, DESC, EXCLUDES, FIRST, FROM, GROUP, HAVING, IN, INCLUDES, LAST, LIKE, LIMIT, NOT, NULL, NULLS, OR, SELECT, WHERE, WITH

SOQL SELECT Syntax

SOQL query syntax consists of a required SELECT statement followed by one or more optional clauses, such as TYPEOF, WHERE, WITH, GROUP BY, and ORDER BY.

The SOQL SELECT statement uses the following syntax:

```
SELECT fieldList [subquery][...]
[TYPEOF typeOfField whenExpression[...] elseExpression END][...]
FROM objectType[,...]
   [USING SCOPE filterScope]
[WHERE conditionExpression]
[WITH [DATA CATEGORY] filteringExpression]
[GROUP BY {fieldGroupByList|ROLLUP (fieldSubtotalGroupByList)|CUBE (fieldSubtotalGroupByList)]
   [HAVING havingConditionExpression] ]
[ORDER BY fieldOrderByList {ASC|DESC} [NULLS {FIRST|LAST}] ]
[LIMIT numberOfRowsToReturn]
[OFFSET numberOfRowsToSkip]
[FOR {VIEW | REFERENCE}[,...] ]
   [ UPDATE {TRACKING|VIEWSTAT}[,...] ]
```

Syntax

Description

fieldList subquery

Specifies a list of one or more fields, separated by commas, that you want to retrieve from the specified <code>object</code>. The bold elements in the following examples are <code>fieldlists</code>:

- SELECT Id, Name, BillingCity FROM Account
- SELECT count() FROM Contact
- SELECT Contact.Firstname, Contact.Account.Name FROM Contact

You must specify valid field names and must have read-level permissions to each specified field. The fieldist defines the ordering of fields in the query results.

Syntax	Description
	If the query traverses a relationship, <code>fieldList</code> can include a subquery. For example:
	SELECT Account.Name, (SELECT Contact.LastName FROM Account.Contacts) FROM Account
	The <i>fieldlist</i> can also be an aggregate function, such as COUNT() and COUNT(<i>fieldName</i>), or be wrapped in Translating Results.
typeOfField	A polymorphic relationship field in objectType or a polymorphic field in a parent of objectType that can reference multiple object types. For example, the Who relationship field of a Task can be a Contact or a Lead. typeOfField cannot reference a relationship field that is also referenced in fieldList . See TYPEOF for more information.
whenExpression	A clause of the form WHEN whenObjectType THEN whenFieldList . You can have one or more whenExpression clauses inside a TYPEOF expression. See TYPEOF for more information.
elseExpression	A clause of the form ELSE elseFieldList . This clause is optional inside a TYPEOF expression. See TYPEOF for more information.
objectType	Specifies the type of object that you want to query (). You must specify a valid object, such as Account, and must have read-level permissions to that object.
filterScope	Available in API version 32.0 and later. Specifies the filterScope for limiting the results of the query.
conditionExpression	If WHERE is specified, determines which rows and values in the specified object $(objectType)$ to filter against. If unspecified, the query() retrieves all the rows in the object that are visible to the user.
filteringExpression	If WITH DATA CATEGORY is specified, the query() only returns matching records that are associated with the specified data categories and are visible to the user. If unspecified, the query() returns the matching records that are visible to the user. The WITH DATA CATEGORY clause only filters objects of type:
	• Question—to query questions.
	KnowledgeArticleVersion—to query articles.
	For more information about the WITH DATA CATEGORY clause, see WITH DATA CATEGORY filteringExpression.
fieldGroupByList	Available in API version 18.0 and later. Specifies a list of one or more fields, separated by commas, that are used to group the query results. A GROUP BY clause is used with aggregate functions to summarize the data and enable you to roll up query results rather than having to process the individual records in your code. See GROUP BY.
fieldSubtotalGroupByList	Available in API version 18.0 and later. Specifies a list of up to three fields, separated by commas, that are used to group the query results. The results include extra subtotal rows for the grouped data. See GROUP BY ROLLUP and GROUP BY CUBE.
havingConditionExpression	Available in API version 18.0 and later. If the query includes a GROUP BY clause, this conditional expression filters the records that the GROUP BY returns. See HAVING .

Syntax	Description
fieldOrderByList	Specifies a list of one or more fields, separated by commas, that are used to order the query results. For example, you can query for contacts and order the results by last name, and then by first name:
	SELECT Id, LastName, FirstName FROM Contact ORDER BY LastName, FirstName

Note the following implementation tips:

• **Statement Character Limit**—By default, SOQL statements cannot exceed 20,000 characters in length. For SOQL statements that exceed this maximum length, the API returns a MALFORMED QUERY exception code; no result rows are returned.



Note: Long, complex SOQL statements, such as statements that contain many formula fields, can sometimes result in a QUERY_TOO_COMPLICATED error. The error occurs because the statement is expanded internally when processed by Salesforce, even though the original SOQL statement is under the 20,000 character limit. To avoid this error, reduce the complexity of your SOQL statement.

Page layouts in Lightning with more than 250 fields can also cause a QUERY_TOO_COMPLICATED error. Lightning uses auto-generated SOQL to retrieve fields for a record page layout, so the error can occur even if there isn't any customer-written SOQL.

The character limit can also be reached by including too many currency fields. Currency fields require SOQL to use a format method, roughly doubling the field API name length for each currency field.

- **Localized Results**—SELECT statements can include the Translating Results, convertCurrency(), and FORMAT() functions in support of localized fields.
- **Dynamic SOQL in Apex**—Apex requires that you surround SOQL and SOSL statements with square brackets to use them on the fly. You can use Apex script variables and expressions when preceded by a colon (:).
- Ordered Results—The order of results is not guaranteed unless you use an ORDER BY clause in a query.
- **Picklist Values**—In API version 39.0 and later, query for picklist values by the value's API name, which might differ from the actual value.

Condition Expression Syntax (WHERE Clause)

The syntax of the condition expression in a WHERE clause of a SOQL query includes one or more field expressions. You can specify multiple field expressions to a condition expression by using logical operators.

The conditionExpression in the WHERE clause in a SOQL statement uses the following syntax:

```
fieldExpression [logicalOperator fieldExpression2][...]
```

You can add multiple field expressions to a condition expression by using logical operators.

The condition expressions in SOQL SELECT statements appear in bold in these examples:

- SELECT Name FROM Account WHERE Name LIKE 'A%'
- SELECT Id FROM Contact WHERE Name LIKE 'A%' AND MailingState='California'

You can use date or datetime values, or date literals. The format for date and dateTime fields are different.

SELECT Name FROM Account WHERE CreatedDate > 2011-04-26T10:00:00-08:00

SELECT Amount FROM Opportunity WHERE CALENDAR YEAR (CreatedDate) = 2011

For more information on date functions, such as CALENDAR YEAR(), see Date Functions.

You can use parentheses to define the order in which fieldExpressions are evaluated. For example, the following expression is true if fieldExpression1 is true and either fieldExpression2 or fieldExpression3 are true:

```
fieldExpression1 AND (fieldExpression2 OR fieldExpression3)
```

However, the following expression is true if either fieldExpression3 is true or both fieldExpression1 and fieldExpression2 are true.

```
(fieldExpression1 AND fieldExpression2) OR fieldExpression3
```

Client applications must specify parentheses when nesting operators. However, multiple operators of the same type do not need to be nested.



Note: WHERE clauses cannot exceed 4,000 characters.

Using null in SOQL Queries

You can search for null values by using the null keyword.

Use null to represent null values in SOQL queries.

For example, the following statement would return the account IDs of all events with a non-null activity date:

```
SELECT AccountId
FROM Event
WHERE ActivityDate != null
```



Note: The WHERE clause behaves in two different ways, depending on the version, when handling null values in a parent field for a relationship query. In a WHERE clause that checks for a value in a parent field, if the parent does not exist, the record is returned in version 13.0 and later but is not returned in versions before 13.0

```
SELECT Id
FROM Case
WHERE Contact.LastName = null
```

Case record Id values are returned in version 13.0 and later, but are not returned in versions before 13.0.

Translating Results

Use to Label (fields) to translate SOQL guery results into the user's language.

A client application can have results from a query returned that are translated into the user's language, using tolabel():

```
toLabel(object.field)
```

Any organization can use tolabel(). It is particularly useful for organizations that have the Translation Workbench enabled. Use toLabel() to return translations on:

- Regular, multi-select, division, or currency code picklist fields that have values returned by the relevant describe call.
- Data category group and data category unique name fields.
- RecordType names.



Note: Object history is supported on a limited basis. When a tracked field changes, SELECT toLabel (Field) FROM *History returns either the changed field's master label or the translated label, if one exists. OldValue and NewValue fields in field history objects also support toLabel, but only tracked changes on supported field types return translated values.

For example:

```
SELECT Company, toLabel(Recordtype.Name) FROM Lead
```

This query returns lead records with the record type name translated into the language for the user who issued the query.



Note: You can't filter on the translated name value from a record type. Always filter on the master value or the ID of the object for record types.

You can use to Label () to filter records using a translated picklist value. For example:

```
SELECT Company, toLabel(Status)
FROM Lead
WHERE toLabel(Status) = 'le Draft'
```

Lead records are returned where the picklist value for Status is 'le Draft.' The comparison is made against the value for the user's language. If no translation is available for the user's language for the specified picklist, the comparison is made against the master values.



Note: The tolabel() method can't be used with ORDER BY. Salesforce always uses the picklist's defined order, just like reports. Also, you can't use tolabel() in the WHERE clause for division or currency ISO code picklists.

Filtering on Boolean Fields

You can use the Boolean values TRUE and FALSE in SOQL queries.

To filter on a Boolean field, use the following syntax:

```
WHERE BooleanField = TRUE

WHERE BooleanField = FALSE
```

Querying Multi-Select Picklists

You can search for individual values in multi-select picklists, which are regularly used in client applications.

Client applications use a specific syntax for querying multi-select picklists (in which multiple items can be selected). In API version 39.0 and later, query for picklist values by the value's API name, which might differ from the actual value.

The following operators are supported for querying multi-select picklists:

Operator	Description
=	Equals the specified string.
!=	Does not equal the specified string.
includes	Includes (contains) the specified string.
excludes	Excludes (does not contain) the specified string.

Description	Operator	
Specifies AND for two or more strings. Use ; for multi-select picklists when two or more items must be selected. For example:	;	
'AAA;BBB'		
'AAA;BBB'		

Examples

The following query filters on values in the MSP1 c field that are equal to AAA and BBB selected (exact match):

```
SELECT Id, MSP1__c FROM CustObj__c WHERE MSP1__c = 'AAA;BBB'
```

In the following query:

```
SELECT Id, MSP1_c from CustObj_c WHERE MSP1_c includes ('AAA;BBB','CCC')
```

the query filters on values in the MSP1 c field that contains either of these values:

- AAA and BBB selected.
- ccc selected.

A match will result on any field value that contains 'AAA' and 'BBB' or any field that contains 'CCC'. For example, the following will be matched:

matches with 'AAA; BBB':

```
'AAA;BBB'
'AAA;BBB;DDD'
```

matches with ' CCC':

```
'CCC'
'CCC; EEE'
'AAA; CCC'
```

Filtering on Polymorphic Relationship Fields

You can search polymorphic relationship fields on a SQL query. A polymorphic relationship is one where the current object can be one of several object types.

To filter on a polymorphic relationship field, use the Type qualifier.

WHERE polymorphicRelationship. Type comparisonExpression

Syntax	Description
polymorphicRelationship	A polymorphic relationship field in object being queried that can reference multiple object types. For example, the Who relationship field of a Task can be a Contact or a Lead.
comparisonExpression	The comparison being made against the object type in the polymorphic relationship. For more information, see <i>fieldExpression</i> Syntax. Note that type names returned by Type are string values, like 'User'.

The following example only returns records where the What field of Event is referencing an Account or Opportunity. If an Event record referenced a Campaign in the What field, it would not be returned as part of this SELECT.

```
SELECT Id
FROM Event
WHERE What.Type IN ('Account', 'Opportunity')
```

See Understanding Relationship Fields and Polymorphic Fields for more details on polymorphic relationships, and more filtering examples.

fieldExpression Syntax

The field expression syntax of the WHERE clause in a SOQL query consists of a field name, a comparison operator, and a value that's used to compare with the value in the field name.

fieldExpression uses the following syntax:

fieldName comparisonOperator value	
------------------------------------	--

where:

Syntax	Description	
The name of a field in the specified object. Use of single or double quotes around the name van error. You must have at least read-level permissions to the field. It can be any field except a area field, encrypted data field, or base64-encoded field. It does not need to be a field in the factorise.		
comparisonOperator	rator Case-insensitive operators that compare values.	
value	A value used to compare with the value in fieldName. You must supply a value whose data type matches the field type of the specified field. You must supply a native value—other field names or calculations are not permitted. If quotes are required (for example, they are not for dates and numbers), use single quotes. Double quotes result in an error.	

Comparison Operators

Comparison operators, such as =,!=, <, >, LIKE, and IN, can be used in the field expression of the WHERE clause in a SELECT statement in a SOQL query. You can also create more complex queries with semi-joins and anti-joins.

The following table lists the <code>comparisonOperator</code> values that are used in <code>fieldExpression</code> syntax. Comparisons on strings are case-sensitive for unique case-sensitive fields and case-insensitive for all other fields.

Operator	Name	Description
=	Equals	Expression is true if the value in the specified $fieldName$ equals the specified $value$ in the expression. String comparisons using the equals operator are case-sensitive for unique case-sensitive fields and case-insensitive for all other fields.
! =	Not equals	Expression is true if the value in the specified fieldName does not equal the specified value.
<	Less than	Expression is true if the value in the specified fieldName is less than the specified value.

Operator	Name	Description
<=	Less or equal	Expression is true if the value in the specified $fieldName$ is less than, or equals, the specified $value$.
>	Greater than	Expression is true if the value in the specified $fieldName$ is greater than the specified $value$.
>=	Greater or equal	Expression is true if the value in the specified $fieldName$ is greater than or equal to the specified $value$.
LIKE	Like	Expression is true if the value in the specified <code>fieldName</code> matches the characters of the text string in the specified <code>value</code> . The <code>LIKE</code> operator in SOQL and SOSL is similar to the <code>LIKE</code> operator in SQL; it provides a mechanism for matching partial text strings and includes support for wildcards.
		• The % and _ wildcards are supported for the LIKE operator.
		The % wildcard matches zero or more characters.
		The wildcard matches exactly one character.
		 The text string in the specified value must be enclosed in single quotes.
		The LIKE operator is supported for string fields only.
		The LIKE operator performs a case-insensitive match, unlike the case-sensitive matching in SQL.
		• The LIKE operator in SOQL and SOSL supports escaping of special characters $\mbox{\$}$ or $\$
		• Don't use the backslash character in a search except to escape a special character.
		For example, the following query matches Appleton, Apple, and Appl, but not Bappl:
		SELECT AccountId, FirstName, lastname FROM Contact
		WHERE lastname LIKE 'appl%'
IN	IN	If the value equals any one of the specified values in a WHERE clause. For example:
		SELECT Name FROM Account WHERE BillingState IN ('California', 'New York')
		The values for IN must be in parentheses. String values must be surrounded by single quotes.
		IN and NOT IN can also be used for semi-joins and anti-joins when querying on ID (primary key) or reference (foreign key) fields.
NOT IN	NOT IN	If the value does not equal any of the specified values in a WHERE clause. For example:
		SELECT Name FROM Account WHERE BillingState NOT IN ('California', 'New York')
		The values for NOT IN must be in parentheses, and string values must be surrounded by single quotes.
		There is also a logical operator NOT, which is unrelated to this comparison operator.

INCLUDES Applies EXCLUDES	only to multi-select picklists.

Semi-Joins with TN and Anti-Joins with NOT TN

You can query values in a field where another field on the same object has a specified set of values, using In. For example:

```
SELECT Name FROM Account
WHERE BillingState IN ('California', 'New York')
```

In addition, you can create more complex queries by replacing the list of values in the IN OR NOT IN clause with a subquery. The subquery can filter by ID (primary key) or reference (foreign key) fields. A semi-join is a subquery on another object in an IN clause to restrict the records returned. An anti-join is a subquery on another object in a NOT IN clause to restrict the records returned.

Sample uses of semi-joins and anti-joins include:

- Get all contacts for accounts that have an opportunity with a particular record type.
- Get all open opportunities for accounts that have active contracts.
- Get all open cases for contacts that are the decision maker on an opportunity.
- Get all accounts that do not have any open opportunities.

If you filter by an ID field, you can create parent-to-child semi- or anti-joins, such as Account to Contact. If you filter by a reference field, you can also create child-to-child semi- or anti-joins, such as Contact to Opportunity, or child-to-parent semi- or anti-joins, such as Opportunity to Account.

ID field Semi-Join

You can include a semi-join in a WHERE clause. For example, the following query returns account IDs if an associated opportunity is lost:

```
SELECT Id, Name
FROM Account
WHERE Id IN
  ( SELECT AccountId
    FROM Opportunity
    WHERE StageName = 'Closed Lost'
)
```

This example is a parent-to-child semi-join from Account to Opportunity. Notice that the left operand, Id, of the IN clause is an ID field. The subquery returns a single field of the same type as the field to which it is compared. A full list of restrictions that prevent unnecessary processing is provided at the end of this section.

Reference Field Semi-Join

The following query returns task IDs for all contacts in Twin Falls:

```
SELECT Id
FROM Task
WHERE Whold IN
  (
    SELECT Id
    FROM Contact
    WHERE MailingCity = 'Twin Falls'
)
```

Notice that the left operand, Whold, of the IN clause is a reference field. An interesting aspect of this query is that Whold is a polymorphic reference field as it can point to a contact or a lead. The subquery restricts the results to contacts.

ID field Anti-Join

The following query returns account IDs for all accounts that do not have any open opportunities:

```
SELECT Id
FROM Account
WHERE Id NOT IN
  (
    SELECT AccountId
    FROM Opportunity
    WHERE IsClosed = false
)
```

Reference Field Anti-Join

The following query returns opportunity IDs for all contacts whose source is not Web:

```
SELECT Id
FROM Opportunity
WHERE AccountId NOT IN
  (
    SELECT AccountId
    FROM Contact
    WHERE LeadSource = 'Web'
)
```

This example is a child-to-child anti-join from Opportunity to Contact.

Multiple Semi-Joins or Anti-Joins

You can combine semi-join or anti-join clauses in a query. For example, the following query returns account IDs that have open opportunities if the last name of the contact associated with the account is like the last name "Apple":

```
SELECT Id, Name
FROM Account
WHERE Id IN
  (
    SELECT AccountId
    FROM Contact
    WHERE LastName LIKE 'apple%'
)
AND Id IN
  (
    SELECT AccountId
    FROM Opportunity
    WHERE isClosed = false
)
```

You can use at most two subqueries in a single semi-join or anti-join query. Multiple semi-joins and anti-join queries are also subject to existing limits on subqueries per query.

Semi-Joins or Anti-Joins Evaluating Relationship Queries

You can create a semi-join or anti-join that evaluates a relationship query in a SELECT clause. For example, the following query returns opportunity IDs and their related line items if the opportunity's line item total value is more than \$10,000:

```
SELECT Id, (SELECT Id from OpportunityLineItems)
FROM Opportunity
```

```
WHERE Id IN
  (
    SELECT OpportunityId
    FROM OpportunityLineItem
    WHERE totalPrice > 10000
)
```

Because a great deal of processing work is required for semi-join and anti-join queries, Salesforce imposes the following restrictions to maintain the best possible performance:

Basic limits:

- No more than two IN or NOT IN statements per WHERE clause.
- You cannot use the NOT operator as a conjunction with semi-joins and anti-joins. Using it converts a semi-join to an anti-join, and the reverse. Instead of using the NOT operator, write the query in the appropriate semi-join or anti-join form.

Main query limits:

The following restrictions apply to the main WHERE clause of a semi-join or anti-join query:

- The left operand must query a single ID (primary key) or reference (foreign key) field. The selected field in a subquery can be a reference field. For example:

```
SELECT Id

FROM Idea

WHERE (Id IN (SELECT ParentId FROM Vote WHERE CreatedDate > LAST_WEEK AND

Parent.Type='Idea'))
```

The left operand can't use relationships. For example, the following semi-join query is invalid due to the Account.Id relationship field:

```
SELECT Id
FROM Contact
WHERE Account.Id IN
(
SELECT ...
)
```

Subquery limits:

- A subquery must query a field referencing the same object type as the main query.
- There is no limit on the number of records matched in a subquery. Standard SOQL query limits apply to the main query.
- The selected column in a subquery must be a foreign key field, and cannot traverse relationships. This limit means that you cannot use dot notation in a selected field of a subquery. For example, the following query is valid:

```
SELECT Id, Name
FROM Account
WHERE Id IN

(
SELECT AccountId
FROM Contact
WHERE LastName LIKE 'Brown_%'
)
```

Using Account. Id (dot notation) instead of AccountId is not supported. Similarly, subqueries like Contact. AccountId FROM Case are invalid.

 You cannot query on the same object in a subquery as in the main query. You can write such self semi-join queries without using semi-joins or anti-joins. For example, the following self semi-join query is invalid:

```
SELECT Id, Name
FROM Account
WHERE Id IN
  (
    SELECT ParentId
    FROM Account
    WHERE Name = 'myaccount'
)
```

However, it is simple to rewrite the query in a valid form, for example:

```
SELECT Id, Name
FROM Account
WHERE Parent.Name = 'myaccount'
```

- You cannot nest a semi-join or anti-join statement in another semi-join or anti-join statement.
- You can use semi-joins and anti-joins in the main WHERE statement, but not in a subquery WHERE statement. For example, the following query is valid:

The following query is invalid since the nested query is an extra level deep:

- You cannot use subqueries with OR.
- COUNT, FOR UPDATE, ORDER BY, and LIMIT are not supported in subqueries.
- The following objects are not currently supported in subqueries:
 - ActivityHistory
 - Attachments
 - Event
 - EventAttendee
 - Note
 - OpenActivity

- Tags (AccountTag, ContactTag, and all other tag objects)
- Task

Logical Operators

Logical operators can be used in the field expression of the WHERE clause in a SOQL query. These operators are AND, OR, and NOT. The following table lists the logical operator values that are used in fieldExpression syntax:

Operator	Syntax	Description
AND	<pre>fieldExpressionX AND fieldExpressionY</pre>	true if both fieldExpressionX and fieldExpressionY are true.
OR fieldExpressionX Of fieldExpressionY	fieldExpressionX OR fieldExpressionY	true if either <code>fieldExpressionX</code> or <code>fieldExpressionY</code> is true. Relationship queries with foreign key values in an OR clause behave differently depending on the version of the API. In a <code>WHERE</code> clause that uses OR, if the foreign key value in a record is null, the record is returned in version 13.0 and later, but not returned in versions before 13.0.
		<pre>SELECT Id FROM Contact WHERE LastName = 'foo' or Account.Name = 'bar'</pre>
		The contact with no parent account has a last name that meets the criteria, so it is returned in version 13.0 and later.
NOT	not fieldExpressionX	true if <i>fieldExpressionX</i> is false. There is also a comparison operator NOT IN, which is different from this logical operator.

Date Formats and Date Literals

In a SOQL query you can specify either a particular date or a date literal. A date literal is a fixed expression that represents a relative range of time, such as last month, this week, or next year.

dateTime field values are stored as Coordinated Universal Time (UTC). When a dateTime value is returned in Salesforce, it's adjusted for the time zone specified in your org preferences. SOQL queries, however, return dateTime field values as UTC values. If you want to process these values in different time zones, your application might need to handle the conversion.

Date Formats

A fieldExpression uses different date formats for date and dateTime fields. If you specify a dateTime format in a query, you can filter only on dateTime fields. Similarly, if you specify a date format value, you can filter only on date fields.:

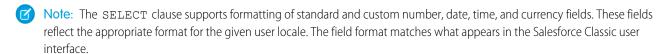
Format	Format Syntax	Example
Date only	YYYY-MM-DD	1999-01-01
Date, time, and time zone	• YYYY-MM-DDThh:mm:ss+hh:mm	• 1999-01-01T23:01:01+01:00
offset	• YYYY-MM-DDThh:mm:ss-hh:mm	• 1999-01-01T23:01:01-08:00

Format	Format Syntax	Example
	• YYYY-MM-DDThh:mm:ssZ	• 1999-01-01T23:01:01Z

The zone offset is always from UTC. For more information, see:

- http://www.w3.org/TR/xmlschema-2/#isoformats
- http://www.w3.org/TR/NOTE-datetime
- Note: For a fieldExpression that uses date formats, the date is not enclosed in single quotes. Don't use quotes around the date. For example:

SELECT Id
FROM Account
WHERE CreatedDate > 2005-10-08T01:02:03Z



Date Literals

A fieldExpression can use a date literal to compare a range of values to the value in a date or dateTime field. Each literal is a range of time beginning with midnight (00:00:00). To find a value within the range, use =. To find values on either side of the range, use > or <. The following table shows the available list of date literals, the ranges they represent, and examples.

Date Literal	Range	Example
YESTERDAY	Starts 00:00:00 the day before and continues for 24 hours.	SELECT Id FROM Account WHERE CreatedDate = YESTERDAY
TODAY	Starts 00:00:00 of the current day and continues for 24 hours.	SELECT Id FROM Account WHERE CreatedDate > TODAY
TOMORROW	Starts 00:00:00 after the current day and continues for 24 hours.	SELECT Id FROM Opportunity WHERE CloseDate = TOMORROW
LAST_WEEK	Starts 00:00:00 on the first day of the week before the most recent first day of the week and continues for seven full days. Your locale determines the first day of the week.	SELECT Id FROM Account WHERE CreatedDate > LAST_WEEK
THIS_WEEK	Starts 00:00:00 on the most recent first day of the week before the current day and continues for seven full days. Your locale determines the first day of the week.	SELECT Id FROM Account WHERE CreatedDate < THIS_WEEK
NEXT_WEEK	Starts 00:00:00 on the most recent first day of the week after the current day and continues for seven full days. Your locale determines the first day of the week.	SELECT Id FROM Opportunity WHERE CloseDate = NEXT_WEEK

Date Literal	Range	Example
LAST_MONTH	Starts 00:00:00 on the first day of the month before the current day and continues for all the days of that month.	SELECT Id FROM Opportunity WHERE CloseDate > LAST_MONTH
THIS_MONTH	Starts 00:00:00 on the first day of the month that the current day is in and continues for all the days of that month.	
NEXT_MONTH	Starts 00:00:00 on the first day of the month after the month that the current day is in and continues for all the days of that month.	SELECT Id FROM Opportunity WHERE CloseDate = NEXT_MONTH
LAST_90_DAYS	Starts 00:00:00 of the current day and continues for the past 90 days.	SELECT Id FROM Account WHERE CreatedDate = LAST_90_DAYS
NEXT_90_DAYS	Starts 00:00:00 of the current day and continues for the next 90 days.	SELECT Id FROM Opportunity WHERE CloseDate > NEXT_90_DAYS
LAST_N_DAYS: n	For the number n provided, starts 00:00:00 of the current day and continues for the past n days.	SELECT Id FROM Account WHERE CreatedDate = LAST_N_DAYS:365
NEXT_N_DAYS:n	For the number n provided, starts 00:00:00 of the current day and continues for the next n days.	SELECT Id FROM Opportunity WHERE CloseDate > NEXT_N_DAYS:15
NEXT_N_WEEKS: n	For the number n provided, starts 00:00:00 of the first day of the next week and continues for the next n weeks.	SELECT Id FROM Opportunity WHERE CloseDate > NEXT_N_WEEKS:4
LAST_N_WEEKS:n	For the number n provided, starts 00:00:00 of the last day of the previous week and continues for the past n weeks.	<pre>SELECT Id FROM Account WHERE CreatedDate = LAST_N_WEEKS:52</pre>
NEXT_N_MONTHS:n	For the number n provided, starts 00:00:00 of the first day of the next month and continues for the next n months.	SELECT Id FROM Opportunity WHERE CloseDate > NEXT_N_MONTHS:2
LAST_N_MONTHS:n	For the number n provided, starts 00:00:00 of the last day of the previous month and continues for the past n months.	
THIS_QUARTER	Starts 00:00:00 of the current quarter and continues to the end of the current quarter.	SELECT Id FROM Account WHERE CreatedDate = THIS_QUARTER
LAST_QUARTER	Starts 00:00:00 of the previous quarter and continues to the end of that quarter.	SELECT Id FROM Account WHERE CreatedDate > LAST_QUARTER
NEXT_QUARTER	Starts 00:00:00 of the next quarter and continues to the end of that quarter.	SELECT Id FROM Account WHERE CreatedDate < NEXT_QUARTER
NEXT_N_QUARTERS: n	Starts 00:00:00 of the next quarter and continues to the end of the n th quarter.	SELECT Id FROM Account WHERE CreatedDate < NEXT_N_QUARTERS:2

Date Literal	Range	Example
LAST_N_QUARTERS: n	Starts 00:00:00 of the previous quarter and continues to the end of the previous <i>n</i> th quarter.	SELECT Id FROM Account WHERE CreatedDate > LAST_N_QUARTERS:2
THIS_YEAR	Starts 00:00:00 on January 1 of the current year and continues through the end of December 31 of the current year.	SELECT Id FROM Opportunity WHERE CloseDate = THIS_YEAR
LAST_YEAR	Starts 00:00:00 on January 1 of the previous year and continues through the end of December 31 of that year.	SELECT Id FROM Opportunity WHERE CloseDate > LAST_YEAR
NEXT_YEAR	Starts 00:00:00 on January 1 of the following year and continues through the end of December 31 of that year.	SELECT Id FROM Opportunity WHERE CloseDate < NEXT_YEAR
NEXT_N_YEARS:n	Starts 00:00:00 on January 1 of the following year and continues through the end of December 31 of the <i>n</i> th year.	SELECT Id FROM Opportunity WHERE CloseDate < NEXT_N_YEARS:5
LAST_N_YEARS:n	Starts 00:00:00 on January 1 of the previous year and continues through the end of December 31 of the previous nth year.	SELECT Id FROM Opportunity WHERE CloseDate > LAST_N_YEARS:5
THIS_FISCAL_QUARTER	Starts 00:00:00 on the first day of the current fiscal quarter and continues through the end of the last day of the fiscal quarter. The fiscal year is defined on the Fiscal Year page in Setup.	SELECT Id FROM Account WHERE CreatedDate = THIS_FISCAL_QUARTER
LAST_FISCAL_QUARTER	Starts 00:00:00 on the first day of the last fiscal quarter and continues through the end of the last day of that fiscal quarter. The fiscal year is defined on the Fiscal Year page in Setup.	SELECT Id FROM Account WHERE CreatedDate > LAST_FISCAL_QUARTER
NEXT_FISCAL_QUARTER	Starts 00:00:00 on the first day of the next fiscal quarter and continues through the end of the last day of that fiscal quarter. The fiscal year is defined on the Fiscal Year page in Setup.	SELECT Id FROM Account WHERE CreatedDate < NEXT_FISCAL_QUARTER
NEXT_N_FISCAL_ QUARTERS:n	Starts 00:00:00 on the first day of the next fiscal quarter and continues through the end of the last day of the nth fiscal quarter. The fiscal year is defined on the Fiscal Year page in Setup.	SELECT Id FROM Account WHERE CreatedDate < NEXT_N_FISCAL_QUARTERS:6
LAST_N_FISCAL_ QUARTERS: n	Starts 00:00:00 on the first day of the last fiscal quarter and continues through the end of the last day of the previous <i>n</i> th fiscal quarter. The fiscal year is defined on the Fiscal Year page in Setup.	SELECT Id FROM Account WHERE CreatedDate > LAST_N_FISCAL_QUARTERS:6
THIS_FISCAL_YEAR	Starts 00:00:00 on the first day of the current fiscal year and continues through the end of the last day of the fiscal year. The fiscal year is defined on the Fiscal Year page in Setup.	SELECT Id FROM Opportunity WHERE CloseDate = THIS_FISCAL_YEAR

Date Literal	Range	Example
LAST_FISCAL_YEAR	Starts 00:00:00 on the first day of the last fiscal year and continues through the end of the last day of that fiscal year. The fiscal year is defined on the Fiscal Year page in Setup.	SELECT Id FROM Opportunity WHERE CloseDate > LAST_FISCAL_YEAR
NEXT_FISCAL_YEAR	Starts 00:00:00 on the first day of the next fiscal year and continues through the end of the last day of that fiscal year. The fiscal year is defined on the Fiscal Year page in Setup.	SELECT Id FROM Opportunity WHERE CloseDate < NEXT_FISCAL_YEAR
NEXT_N_FISCAL_ YEARS: n	Starts 00:00:00 on the first day of the next fiscal year and continues through the end of the last day of the <i>n</i> th fiscal year. The fiscal year is defined on the Fiscal Year page in Setup.	
LAST_N_FISCAL_ YEARS: n	Starts 00:00:00 on the first day of the last fiscal year and continues through the end of the last day of the previous <i>n</i> th fiscal year. The fiscal year is defined on the Fiscal Year page in Setup.	SELECT Id FROM Opportunity WHERE CloseDate > LAST_N_FISCAL_YEARS:3



Note: If you've defined custom fiscal years in the Salesforce user interface and in any FISCAL date literals that you specify a range that is outside the years you've defined, an invalid date error is returned.

Minimum and Maximum Dates

Only dates within a certain range are valid. The earliest valid date is 1700-01-01T00:00:00Z GMT, or just after midnight on January 1, 1700. The latest valid date is 4000-12-31T00:00:00Z GMT, or just after midnight on December 31, 4000. These values are offset by your time zone. For example, in the Pacific time zone, the earliest valid date is 1699-12-31T16:00:00, or 4:00 PM on December 31, 1699.

USING SCOPE

The optional USING SCOPE clause of a SOQL query returns records within a specified scope. For example, you can limit the records to return only objects that the user owns or only records in the user's territory.

With API version 32.0 and later, you can use USING SCOPE to limit the results of a query to a specified filterScope. The syntax is:

[USING SCOPE filterScope]

filterScope can take one of the following enumeration values:

Delegated

Filter for records delegated to another user for action. For example, a guery could filter for only delegated Task records.

Everything

Filter for all records.

Mine

Filter for records owned by the user running the query.

MineAndMyGroups

Filter for records assigned to the user running the guery and the user's queues.

My Territory

Filter for records in the territory of the user running the query. This option is available if territory management is enabled for your organization.

My Team Territory

Filter for records in the territory of the team of the user running the query. This option is available if territory management is enabled for your organization.

Team

Filter for records assigned to a team, such as an Account team.



Note: filterScope in SOQL is different from filterScope used in Metadata API.

ORDER BY

Use the optional ORDER BY in a SELECT statement of a SOQL query to control the order of the query results, such as alphabetically beginning with z. If records are null, you can use ORDER BY to display the empty records first or last.

You can use ORDER BY in a SELECT statement to control the order of the query results. There is no guarantee of the order of results unless you use an ORDER BY clause in a query. The syntax is:

[ORDER BY fieldOrderByList {ASC|DESC} [NULLS {FIRST|LAST}]]

Syntax	Description
ASC or DESC	Specifies whether the results are ordered in ascending (ASC) or descending (DESC) order. Default order is ascending.
NULLS FIRST OF NULLS LAST	Orders null records at the beginning (NULLS FIRST) or end (NULLS LAST) of the results. By default, null values are sorted first.

For example, the following query returns a query result with Account records in alphabetical order by first name, sorted in descending order, with accounts that have null names appearing last:

SELECT Name
FROM Account
ORDER BY Name DESC NULLS LAST

The following factors affect results returned with ORDER BY:

- Sorting is case insensitive.
- ORDER BY is compatible with relationship query syntax.
- Multiple column sorting is supported, by listing more than one fieldExpression clause.

• Relationship queries with foreign key values in an ORDER BY clause behave differently depending on the version of the Lightning Platform API. In an ORDER BY clause, if the foreign key value in a record is null, the record is returned in version 13.0 and later, but not returned in versions before 13.0.

```
SELECT Id, CaseNumber, Account.Id, Account.Name
FROM Case
ORDER BY Account.Name
```

Any case record for which AccountId is empty is returned in version 13.0 and later.

• Sort order is determined by current user locale. For English locales, Salesforce uses a sorting mechanism based on the UTF-8 values of the character data. For Asian locales, Salesforce uses a linguistic sorting mechanism based on the ISO 14651 and Unicode 3.2 standards.

The following limitations apply to data types when using ORDER BY:

- These data types are not supported: multi-select picklist, rich text area, long text area, encrypted (if enabled), and data category group reference (if Salesforce Knowledge is enabled).
- All other data types are supported, with the following caveats:
 - Corporate currency always sorts using corporate currency value, if available.
 - phone data does not include any special formatting when sorting, for example, non-numeric characters such as dash or
 parentheses are included in the sorting.
 - picklist sorting is defined by the picklist sort determined during setup.

External objects have the following limitations for the ORDER BY clause.

- The following limits apply only to the OData 2.0 and 4.0 adapters for Salesforce Connect.
 - NULLS FIRST and NULLS LAST are ignored.
 - External objects don't support the ORDER BY clause in relationship queries.
- The following limits apply only to custom adapters for Salesforce Connect.
 - NULLS FIRST and NULLS LAST are ignored.

You can use ORDER BY with the optional LIMIT qualifier, in a SELECT statement:

```
SELECT Name
FROM Account
WHERE industry = 'media'
ORDER BY BillingPostalCode ASC NULLS LAST LIMIT 125
```

LIMIT

LIMIT is an optional clause that can be added to a SELECT statement of a SOQL query to specify the maximum number of rows to return

The syntax for LIMIT is:

```
SELECT fieldList

FROM objectType
[WHERE conditionExpression]
[LIMIT numberOfRows]
```

For example:

```
SELECT Name
FROM Account
WHERE Industry = 'Media' LIMIT 125
```

This query returns the first 125 Account records whose Industry is Media.

You can use LIMIT with count () as the fieldList to count up to the maximum specified.

You can't use a LIMIT clause in a query that uses an aggregate function, but does not use a GROUP BY clause. For example, the following query is invalid:

```
SELECT MAX(CreatedDate)
FROM Account LIMIT 1
```

OFFSET

When expecting many records in a query's results, you can display the results in multiple pages by using the OFFSET clause on a SOQL query. For example, you can use OFFSET to display records 51–75 and then jump to displaying records 301–350. Using OFFSET is an efficient way to handle large results sets.

Use OFFSET to specify the starting row offset into the result set returned by your query. Because the offset calculation is done on the server and only the result subset is returned, using OFFSET is more efficient than retrieving the full result set and then filtering the results locally. OFFSET is available in API version 24.0 and later.

```
SELECT fieldList

FROM objectType
[WHERE conditionExpression]

ORDER BY fieldOrderByList

LIMIT numberOfRowsToReturn

OFFSET numberOfRowsToSkip
```

As an example, if a SOQL query normally returned 50 rows, you could use OFFSET 10 in your query to skip the first 10 rows:

```
SELECT Name

FROM Merchandise__c

WHERE Price__c > 5.0

ORDER BY Name

LIMIT 100

OFFSET 10
```

The result set for the preceding example would be a subset of the full result set, returning rows 11 through 110 of the full set.

Considerations When Using OFFSET

Here are a few points to consider when using OFFSET in your queries:

- The maximum offset is 2,000 rows. Requesting an offset greater than 2,000 results in a NUMBER_OUTSIDE_VALID_RANGE error.
- OFFSET is intended to be used in a top-level query, and is not allowed in most subqueries, so the following query is invalid and returns a MALFORMED_QUERY error:

```
SELECT Name, Id
FROM Merchandise__c
```

```
WHERE Id IN

(

SELECT Id

FROM Discontinued_Merchandise_c

LIMIT 100

OFFSET 20

)

ORDER BY Name
```

A subquery can use OFFSET only if the parent query has a LIMIT 1 clause. The following query is a valid use of OFFSET in a subquery:

```
SELECT Name, Id

(
SELECT Name FROM Opportunity LIMIT 10 OFFSET 2
)
FROM Account
ORDER BY Name
LIMIT 1
```

OFFSET cannot be used as a subquery in the WHERE clause, even if the parent query uses LIMIT 1.

- Note: Using OFFSET in subqueries is a pilot feature that is subject to change in future releases, and is not intended for use in a production setting. There is no support associated with this pilot feature. For more information, contact Salesforce
- We recommend using an ORDER BY clause when you use OFFSET to ensure that the result set ordering is consistent. The row order of a result set that does not have an ORDER BY clause has a stable ordering, however the ordering key is subject to change and should not be relied on.
- Similarly, we recommend using a LIMIT clause with OFFSET if you need to retrieve subsequent subsets of the same result set. For example, you could retrieve the first 100 rows of a query using the following:

```
SELECT Name, Id
FROM Merchandise__c
ORDER BY Name
LIMIT 100
OFFSET 0
```

You could then retrieve the next 100 rows, 101 through 201, using the following query:

```
SELECT Name, Id
FROM Merchandise__c
ORDER BY Name
LIMIT 100
OFFSET 100
```

• OFFSET is applied to the result set returned at the time of the query. No server-side cursor is created to cache the full result set for future OFFSET queries. The page results may change if the underlying data is modified during multiple queries using OFFSET into the same result set. As an example, suppose that the following query normally returns a full result set of 50 rows, and the first 10 rows are skipped using an OFFSET clause:

```
SELECT Name
FROM Merchandise__c
ORDER BY Name
OFFSET 10
```

After the query is run, 10 new rows are then inserted into Merchandise__c with Name values that come early in the sort order. If the query is run again, with the same OFFSET value, a different set of rows is skipped. If you need to query multiple pages of records with a consistent server-side cursor, use the queryMore () in SOAP API.

- Offsets are not intended to be used as a replacement for using queryMore(), given the maximum offset size and lack of a server-side cursor. Multiple queries with offsets into a large result set has a higher performance impact than using queryMore() against a server-side cursor.
- When using OFFSET, only the first batch of records are returned for a given query. If you want to retrieve the next batch, you'll need to re-execute the query with a higher offset value.
- The OFFSET clause is allowed in SOQL used in SOAP API, REST API, and Apex. It's not allowed in SOQL used within Bulk API or Streaming API.

Update an Article's Keyword Tracking with SOQL

Track keywords that are used in Salesforce Knowledge article searches with the UPDATE TRACKING optional clause on a SOQL query.

UPDATE TRACKING is an optional clause that can be added to a SELECT statement of a SOQL query to report on article searches and views. Developers can use UPDATE TRACKING to track the keywords that are used in Salesforce Knowledge article searches.



Example: You can use this syntax to track a keyword that are used in Salesforce Knowledge article search:

```
SELECT Title FROM FAQ__kav

WHERE Keyword='Apex' and

Language = 'en_US' and

KnowledgeArticleVersion = 'ka23000000PCiy'

UPDATE TRACKING
```

Update an Article Viewstat with SOQL

Determine how many hits a Salesforce Knowledge article has had by using the UPDATE VIEWSTAT optional clause on a SOQL query. You can get a view count for every article that you have access to online.

The UPDATE VIEWSTAT clause is used in a SELECT statement to report on Salesforce Knowledge article searches and views. It allows developers to update an article's view statistics.

You can use this syntax to increase the view count for every article you have access to online:

```
SELECT Title FROM FAQ__kav
WHERE PublishStatus='online' and
Language = 'en_US' and
KnowledgeArticleVersion = 'ka23000000PCiy'
UPDATE VIEWSTAT
```

with filteringExpression

You can filter records based on field values, for example, to filter according to category or to query and retrieve changes that are tracked in a user's profile feed by using WITHfilteringExpression. This optional clause can be added to a SELECT statement of a SOQL query.

Unlike the WHERE clause which only supports fields from the object specified in the FROM clause, WITH allows you to filter by other related criteria. For example, you can use the WITH clause to filter articles based on their classification in one or more data category groups. The WITH clause can only be used in the following cases:

- To filter records based on their categorization. See WITH DATA CATEGORY filteringExpression.
- To query and retrieve record changes tracked in a user profile feed. See Custom Object_Feed in the Object Reference for Salesforce
- To filter records based on user access permissions on fields or objects being queried within Apex code. See Filter SOQL Queries Using WITH SECURITY_ENFORCED (Beta) in the *Apex Developer Guide*.

If WITH is specified, the query returns only records that match the filter and are visible to the user. If unspecified, the query returns only the matching records that are visible to the user.

The filtering expression in this statement is highlighted in bold. The syntax is explained in the following sections.

- SELECT Title FROM KnowledgeArticleVersion WHERE PublishStatus='online' WITH DATA
 CATEGORY Geography c ABOVE usa c
- SELECT Id FROM UserProfileFeed WITH UserId='005D000001AamR' ORDER BY CreatedDate DESC, Id DESC LIMIT 20

WITH DATA CATEGORY filteringExpression

You can search for Salesforce Knowledge articles and questions by their data category in a SOQL query. WITH DATA CATEGORY is an optional clause in a SELECT statement that's used to filter records that are associated with one or more data categories and are visible to users.

If WITH DATA CATEGORY is specified, the query () returns only matching records that are associated with the specified data categories and are visible to the user. If unspecified, the query () only returns the matching records that are visible to the user.

(1) Important: CategoryData is an object and DATA CATEGORY is syntax in a SOQL WITH clause. WITH DATA CATEGORY is valid syntax, but WITH CategoryData is not supported.

A SOQL statement using a WITH DATA CATEGORY clause must also include a FROM **ObjectTypeName** clause where **ObjectTypeName** equals:

- KnowledgeArticleVersion to query all article types
- an article type API Name to query a specific article type
- Question to query questions

When ObjectTypeName equals to KnowledgeArticleVersion or any article type API Name in the FROM clause, a WHERE clause must be specified with one of the following parameters:

- PublishStatus to query articles depending on their status in the publishing cycle:
 - WHERE PublishStatus='online' for published articles
 - WHERE PublishStatus='archived' for archived articles
 - WHERE PublishStatus='draft' for draft articles
- Id to query an article based on its id

For information on article types or questions, see "Knowledge Article Types" and "Find and View Questions" in the Salesforce Help.

✓ Note: The WITH DATA CATEGORY clause does not support bind variables.

filteringExpression

The filteringExpression in the WITH DATA CATEGORY clause uses the following syntax:

dataCategorySelection [AND [dataCategorySelection2][...]

The examples in this section are based on the following data category group:

```
Geography__c
ww__c
northAmerica__c
usa__c
canada__c
mexico__c
europe__c
france__c
uk__c
asia__c
```

The category filtering in the statements below is highlighted in bold. The syntax is explained in the following sections.

- SELECT Title FROM KnowledgeArticleVersion WHERE PublishStatus='online' WITH DATA CATEGORY Geography_c ABOVE usa_c
- SELECT Title FROM Question WHERE LastReplyDate > 2005-10-08T01:02:03Z WITH DATA CATEGORY Geography c AT (usa c, uk c)
- SELECT UrlName FROM KnowledgeArticleVersion WHERE PublishStatus='draft' WITH DATA CATEGORY Geography c AT usa c AND Product c ABOVE OR BELOW mobile phones c

You can only use the AND logical operator. The following syntax is incorrect as OR is not supported:

```
WITH DATA CATEGORY Geography_c ABOVE usa_c OR Product_c AT mobile_phones_c
```

dataCategorySelection

The syntax of the data category selection in a WITH DATA CATEGORY clause in a SOQL query includes a category group name to use as a filter, the filter selector, and the name of the category to use for filtering.

The dataCategorySelection uses the following syntax:

 $data Category {\it Group Name filtering Selector data Category Name}$

Syntax	Description
dataCategoryGroupName	The name of the data category group to use as a filter. Geography_c is the data category group in the following example.
	You cannot use the same data category group more than once in a query. As an example, the following command is incorrect: WITH DATA CATEGORY Geography_c ABOVE usa_c AND Geography_c BELOW europe_c
filteringSelector	The selector used to filter the data in the specified data category. See Filtering Selectors for a list of valid selectors.
dataCategoryName	The name of the data category for filtering. You must have visibility on the category you specify. For more information on category visibility, see "Data Category Visibility" in the Salesforce Help.
	You can use parentheses to apply the filtering operator to more than one data category. Each data category must be separated by a comma.
	<pre>Example: WITH DATA CATEGORY Geographyc AT (usac,francec,ukc)</pre>

Syntax	Description	
	You can't use the AND operator instead of parentheses to list multiple data categories. The following syntax does not work WITH DATA CATEGORY Geographyc AT usac AND francec	

Filtering Selectors

When specifying filters for a WITH CATEGORY clause of a SOQL query, you can use AT to select the specified category, ABOVE to select the category and all its parent categories, BELOW to select the category and all its subcategories, and ABOVE_OR_BELOW to select the category, its parent categories, and its subcategories.

The following table lists the *filteringSelector* values that are used in the *dataCategorySelection* syntax.

The examples in this section are based on the following data category group:

```
Geography__c
ww__c
northAmerica__c
usa__c
canada__c
mexico__c
europe__c
france__c
uk__c
asia__c
```

Selector	Description
AT	Select the specified data category.
	For example, the following syntax selects asiac.
	WITH DATA CATEGORY Geographyc AT asiac
ABOVE	Select the specified data category and all its parent categories.
	For example, the following syntax selects usac, northAmerica_c, and wwc.
	WITH DATA CATEGORY Geographyc ABOVE usac
BELOW	Select the specified data category and all its subcategories.
	For example the following selects northAmericac, usac, canadac, and mexicoc.
	WITH DATA CATEGORY Geographyc BELOW northAmericac
ABOVE_OR_BELOW	Select the specified data category and:
	all its parent categories
	all its subcategories
	For example the following selects wwc, europec, francec and ukc.
	WITH DATA CATEGORY Geographyc ABOVE_OR_BELOW europec



Note: For more information on data category groups, data categories, parent and subcategories, see "Work with Data Categories" in the Salesforce Help.

Example with data category Clauses

Here are examples of WITH DATA CATEGORY clauses in a SELECT statement in a SOQL query.

Type of Search	Example(s)
Select the title from all questions classified with the mobile_phonesc data category in the Productc data category group	SELECT Title FROM Question WHERE LastReplyDate < 2005-10-08T01:02:03Z WITH DATA CATEGORY Productc AT mobile_phonesc
Select the title and summary from all published Knowledge articles classified: • above or below europec in the Geographyc data category group • below allProducts_c in the Product_c data category group	SELECT Title, Summary FROM KnowledgeArticleVersion WHERE PublishStatus='Online' AND Language = 'en_US' WITH DATA CATEGORY Geography_c ABOVE_OR_BELOW europe_c AND Product_c BELOW All_c
Select the ID and title from draft articles of type "Offer_kav" classified: with the france_c or usa_c data category in the Geography_c data category group above the dsl_c data category group	SELECT Id, Title FROM Offerkav WHERE PublishStatus='Draft' AND Language = 'en_US' WITH DATA CATEGORY Geographyc AT (francec,usac) AND Productc ABOVE dslc

GROUP BY

You can use the GROUP BY option in a SOQL query to avoid iterating through individual query results. That is, you specify a group of records instead of processing many individual records.

With API version 18.0 and later, you can use GROUP BY with aggregate functions, such as SUM() or MAX(), to summarize the data and enable you to roll up query results rather than having to process the individual records in your code. The syntax is:

```
[GROUP BY fieldGroupByList]
```

fieldGroupByList specifies a list of one or more fields, separated by commas, that you want to group by. If the list of fields in a SELECT clause includes an aggregate function, you must include all non-aggregated fields in the GROUP BY clause.

For example, to determine how many leads are associated with each LeadSource value without using GROUP BY, you could run the following query:

```
SELECT LeadSource FROM Lead
```

You would then write some code to iterate through the query results and increment counters for each LeadSource value. You can use GROUP BY to get the same results without the need to write any extra code. For example:

```
SELECT LeadSource, COUNT(Name)
FROM Lead
GROUP BY LeadSource
```

For a list of aggregate functions supported by SOQL, see Aggregate Functions.

You can use a GROUP BY clause without an aggregated function to query all the distinct values, including null, for an object. The following query returns the distinct set of values stored in the LeadSource field.

```
SELECT LeadSource
FROM Lead
GROUP BY LeadSource
```

Note that the COUNT DISTINCT () function returns the number of distinct non-null field values matching the query criteria.

Considerations When Using GROUP BY

When you're creating SOQL queries with the GROUP BY clause, there are some considerations to keep in mind.

- Some object fields have a field type that does not support grouping. You can't include fields with these field types in a GROUP BY clause. The Field object associated with DescribeSObjectResult has a groupable field that defines whether you can include the field in a GROUP BY clause.
- You must use a GROUP BY clause if your query uses a LIMIT clause and an aggregated function. For example, the following
 query is valid:

```
SELECT Name, Max(CreatedDate)
FROM Account
GROUP BY Name
LIMIT 5
```

The following query is invalid as there is no GROUP BY clause:

```
SELECT MAX(CreatedDate)
FROM Account LIMIT 1
```

• You can't use child relationship expressions that use the <u>r</u> syntax in a query that uses a GROUP BY clause. For more information, see Understanding Relationship Names, Custom Objects, and Custom Fields on page 58.

GROUP BY and queryMore()

For queries that don't include a GROUP BY clause, the query result object contains up to 500 rows of data by default. If the query results exceed 500 rows, then your client application can use the queryMore () call and a server-side cursor to retrieve additional rows in 500-row chunks.

However, if a query includes a GROUP BY clause, you can't use queryMore (). You can increase the default size up to 2,000 in the QueryOptions header. If your query results exceed 2,000 rows, you must change the filtering conditions to query data in smaller chunks. There is no guarantee that the requested batch size will be the actual batch size. This is done to maximize performance. See Change the Batch Size in Queries for more details.

GROUP BY and Subtotals

If you want a query to do the work of calculating subtotals so that you don't have to maintain that logic in your code, see GROUP BY ROLLUP. If you want to calculate subtotals for every possible combination of grouped field (to generate a cross-tabular report, for example), see GROUP BY CUBE instead.

Using Aliases with GROUP BY

You can use an alias for any field or aggregated field in a SELECT statement in a SOQL query. Use a field alias to identify the field when you're processing the query results in your code.

Specify the alias directly after the associated field. For example, the following query contains two aliases: n for the Name field, and max for the MAX (Amount) aggregated field.

```
SELECT Name n, MAX(Amount) max
FROM Opportunity
GROUP BY Name
```

Any aggregated field in a SELECT list that does not have an alias automatically gets an implied alias with a format exprile j, where j denotes the order of the aggregated fields with no explicit aliases. The value of j starts at 0 and increments for every aggregated field with no explicit alias.

In the following example, MAX (Amount) has an implied alias of expr0, and MIN (Amount) has an implied alias of expr1.

```
SELECT Name, MAX(Amount), MIN(Amount)
FROM Opportunity
GROUP BY Name
```

In the next query, MIN (Amount) has an explicit alias of min. MAX (Amount) has an implied alias of expr0, and SUM (Amount) has an implied alias of expr1.

```
SELECT Name, MAX(Amount), MIN(Amount) min, SUM(Amount)
FROM Opportunity
GROUP BY Name
```

GROUP BY ROLLUP

Use the GROUP BY ROLLUP optional clause in a SOQL query to add subtotals for aggregated data in query results. This action enables the query to calculate subtotals so that you don't have to maintain that logic in your code.

With API version 18.0 and later, you can use GROUP BY ROLLUP with aggregate functions, such as SUM() and COUNT (*fieldName*). The syntax is:

```
[GROUP BY ROLLUP (fieldName[,...])]
```

A query with a GROUP BY ROLLUP clause returns the same aggregated data as an equivalent query with a GROUP BY clause. It also returns multiple levels of subtotal rows. You can include up to three fields in a comma-separated list in a GROUP BY ROLLUP clause.

The GROUP BY ROLLUP clause adds subtotals at different levels, aggregating from right to left through the list of grouping columns. The order of rollup fields is important. A query that includes three rollup fields returns the following rows for totals:

- First-level subtotals for each combination of fieldName1 and fieldName2. Results are grouped by fieldName3.
- Second-level subtotals for each value of fieldName1. Results are grouped by fieldName2 and fieldName3.
- One grand total row



- You can't combine GROUP BY and GROUP BY ROLLUP syntax in the same statement. For example, GROUP BY ROLLUP (field1), field2 is not valid as all grouped fields must be within the parentheses.
- If you want to compile a cross-tabular report including subtotals for every possible combination of fields in a GROUP BY clause, use GROUP BY CUBE instead.

Grouping By One Rollup Field

This simple example rolls the results up by one field:

```
SELECT LeadSource, COUNT(Name) cnt
FROM Lead
GROUP BY ROLLUP(LeadSource)
```

The following table shows the query results. Note that the aggregated results include an extra row with a null value for LeadSource that gives a grand total for all the groupings. Since there is only one rollup field, there are no other subtotals.

LeadSource	cnt
Web	7
Phone Inquiry	4
Partner Referral	4
Purchased List	7
null	22

Grouping By Two Rollup Fields

This example rolls the results up by two fields:

```
SELECT Status, LeadSource, COUNT(Name) cnt
FROM Lead
GROUP BY ROLLUP(Status, LeadSource)
```

The following table shows the query results. Note the first-level subtotals and grand total rows. The **Comment** column explains each row.

Status	LeadSource	cnt	Comment
Open - Not Contacted	Web	1	One lead with Status = Open - Not Contacted and LeadSource = Web
Open - Not Contacted	Phone Inquiry	1	One lead with Status = Open - Not Contacted and LeadSource = Phone Inquiry
Open - Not Contacted	Purchased List	1	One lead with Status = Open - Not Contacted and LeadSource = Purchased List
Open - Not Contacted	null	3	First-level subtotal for all leads with Status = Open - Not Contacted

Working - Contacted		LeadSource		Comment
Contacted Working - Partner Referral 3 Three leads with Status = Working - Contacted and LeadSource = Partner Referral Working - Purchased List 4 Four leads with Status = Working - Contacted and LeadSource = Purchased List Working - null 12 First-level subtotal for all leads with Status = Working - Contacted Closed - Web 1 One lead with Status = Closed - Converted and LeadSource = Web	=	Web	4	
Contacted Working - Purchased List 4 Four leads with Status = Working - Contacted and LeadSource = Purchased List Working - null 12 First-level subtotal for all leads with Status = Working - Contacted Closed - Web 1 One lead with Status = Closed - Converted and LeadSource = Web		Phone Inquiry	1	
Contacted LeadSource = Purchased List Working - null 12 First-level subtotal for all leads with Status = Working - Contacted Closed - Web 1 One lead with Status = Closed - Converted and LeadSource = Web	=	Partner Referral	3	
Contacted Closed - Web 1 One lead with Status = Closed - Converted and Converted LeadSource = Web	-	Purchased List	4	_
Converted LeadSource = Web	-	null	12	-
Closed - Phone Inquiry 1 One lead with Status = Closed - Converted and		Web	1	
Converted LeadSource = Phone Inquiry		Phone Inquiry	1	
Closed - Purchased List 1 One lead with Status = Closed - Converted and LeadSource = Purchased List		Purchased List	1	
Closed - null 3 First-level subtotal for all leads with Status = Closed - Converted Converted		null	3	
Closed - Not Web 1 One lead with Status = Closed - Not Converted and Converted LeadSource = Web		Web	1	One lead with Status = Closed - Not Converted and LeadSource = Web
Closed - Not Phone Inquiry 1 One lead with Status = Closed - Not Converted and Converted LeadSource = Phone Inquiry		Phone Inquiry	1	One lead with Status = Closed - Not Converted and LeadSource = Phone Inquiry
Closed - Not Partner Referral 1 One lead with Status = Closed - Not Converted and Converted LeadSource = Partner Referral		Partner Referral	1	One lead with Status = Closed - Not Converted and LeadSource = Partner Referral
Closed - Not Purchased List 1 One lead with Status = Closed - Not Converted and Converted LeadSource = Purchased List		Purchased List	1	One lead with Status = Closed - Not Converted and LeadSource = Purchased List
Closed - Not null 4 First-level subtotal for all leads with Status = Closed - Not Converted Converted		null	4	First-level subtotal for all leads with $Status = Closed - Not Converted$
null null 22 Grand total of 22 leads	ull	null	22	Grand total of 22 leads

These examples use the COUNT (**fieldName**) aggregate function, but the syntax works with any aggregate function. You can also group by three rollup fields, which returns even more subtotal rows.

Using GROUPING (fieldName) to Identify Subtotals

You can use the GROUPING (**fieldName**) function to determine whether a row is a subtotal or field when you use GROUP BY ROLLUP or GROUP BY CUBE in SOQL queries.

The GROUP BY ROLLUP or GROUP BY CUBE clause adds the subtotals, and then the GROUPING (**fieldName**) function identifies whether the row is a subtotal for a field.

If you are iterating through the query result to create a report or chart of the data, you have to distinguish between aggregated data and subtotal rows. You can use GROUPING (**fieldName**) to do this. Using GROUPING (**fieldName**) is more important for interpreting your results when you have more than one field in your GROUP BY ROLLUP or GROUP BY CUBE clause. It is the best way to differentiate between aggregated data and subtotals.

GROUPING (**fieldName**) returns 1 if the row is a subtotal for the field, and 0 otherwise. You can use GROUPING (**fieldName**) in SELECT, HAVING, and ORDER BY clauses.

The easiest way to understand more is to look at a query and its results.

```
SELECT LeadSource, Rating,

GROUPING(LeadSource) grpLS, GROUPING(Rating) grpRating,

COUNT(Name) cnt

FROM Lead

GROUP BY ROLLUP(LeadSource, Rating)
```

The query returns subtotals for combinations of the LeadSource and Rating fields. GROUPING (LeadSource) indicates if the row is an aggregated row for the LeadSource field, and GROUPING (Rating) does the same for the Rating field.

The following table shows the query results. The **Comment** column explains each row.

LeadSource	Rating	grpLS	grpRating	cnt	Comment
Web	null	0	0	5	Five leads with LeadSource = Web with no Rating
Web	Hot	0	0	1	One lead with LeadSource = Web with Rating = Hot
Web	Warm	0	0	1	One lead with LeadSource = Web with Rating = Warm
Web	null	0	1	7	Subtotal of seven leads with LeadSource = Web (grpRating = 1 indicates that result is grouped by the Rating field)
Phone Inquiry	null	0	0	4	Four leads with LeadSource = Phone Inquiry with no Rating
Phone Inquiry	null	0	1	4	Subtotal of four leads with LeadSource = Phone Inquiry (grpRating = 1 indicates that result is grouped by the Rating field)
Partner Referral	null	0	0	4	Four leads with LeadSource = Partner Referral with no Rating
Partner Referral	null	0	1	4	Subtotal of four leads with LeadSource = Partner Referral (grpRating = 1 indicates that result is grouped by the Rating field)
Purchased List	null	0	0	7	Seven leads with LeadSource = Purchased List with no Rating
Purchased List	null	0	1	7	Subtotal of seven leads with LeadSource = Purchased List (grpRating = 1 indicates that result is grouped by the Rating field)

LeadSource	Rating gr	rpLS	grpRating	cnt	Comment
null	null 1		1	22	Grand total of 22 leads (grpRating = 1 and grpLS = 1 indicates this is the grand total)



Tip: The order of the fields listed in the GROUP BY ROLLUP clause is important. For example, if you are more interested in getting subtotals for each Rating instead of for each LeadSource, switch the field order to GROUP BY ROLLUP (Rating, LeadSource).

GROUP BY CUBE

Use the GROUP BY CUBE clause in a SOQL query to add subtotals for all combinations of a grouped field in the query results. This action is useful for compiling cross-tabular reports of data. For example, you can create a cross-tabular query to calculate a sum, an average, or another aggregate function and then group the results by two sets of values: one horizontally, the other, vertically.

With API version 18.0 and later, you can use GROUP BY CUBE with aggregate functions, such as SUM() and COUNT (**fieldName**).

The syntax is:

```
[GROUP BY CUBE (fieldName[,...])]
```

A query with a GROUP BY CUBE clause returns the same aggregated data as an equivalent query with a GROUP BY clause. It also returns additional subtotal rows for each combination of fields specified in the comma-separated grouping list, as well as a grand total. You can include up to three fields in a GROUP BY CUBE clause.



Note:

- You can't combine GROUP BY and GROUP BY CUBE syntax in the same statement. For example, GROUP BY CUBE (field1), field2 is not valid as all grouped fields must be within the parentheses.
- If you only want subtotals for a subset of the grouped field combinations, you should use GROUP BY ROLLUP instead.

The following query returns subtotals of accounts for each combination of Type and BillingCountry:

```
SELECT Type, BillingCountry,

GROUPING(Type) grpType, GROUPING(BillingCountry) grpCty,

COUNT(id) accts

FROM Account

GROUP BY CUBE(Type, BillingCountry)

ORDER BY GROUPING(Type), GROUPING(BillingCountry)
```

The following table shows the query results. The query uses ORDER BY GROUPING (Type), GROUPING (BillingCountry) so that the subtotal and grand total rows are returned after the aggregated data rows. This is not necessary, but it can help you when you are iterating through the query results in your code. The **Comment** column explains each row.

Туре	BillingCountry	gpType	grpCty	accts	Comment
Customer - Direct	null	0	0	6	Six accounts with Type = Customer - Direct with BillingCountry = null
Customer - Channel	USA	0	0	1	One account with Type = Customer - Channel with BillingCountry = USA
Customer - Channel	null	0	0	2	Two accounts with Type = Customer - Channel with BillingCountry = null

Туре	BillingCountry	gpType	grpCty	accts	Comment
Customer - Direct	USA	0	0	1	One account with Type = Customer - Direct with BillingCountry = USA
Customer - Channel	France	0	0	1	One account with Type = Customer - Channel with BillingCountry = France
null	USA	0	0	1	One account with Type = null with BillingCountry = USA
Customer - Channel	null	0	1	4	Subtotal of four accounts with Type = Customer - Channel (grpCty = 1 indicates that result is grouped by the BillingCountry field)
Customer - Direct	null	0	1	7	Subtotal of seven accounts with Type = Customer - Direct (grpCty = 1 indicates that result is grouped by the BillingCountry field)
null	null	0	1	1	Subtotal of one account with Type = null (grpCty = 1 indicates that result is grouped by the BillingCountry field)
null	France	1	0	1	Subtotal of one account with BillingCountry = France (grpType = 1 indicates that result is grouped by the Type field)
null	USA	1	0	3	Subtotal of three accounts with $BillingCountry = USA$ ($grpType = 1$ indicates that result is grouped by the $Type$ field)
null	null	1	0	8	Subtotal of eight accounts with $BillingCountry = null$ ($grpType = 1$ indicates that result is grouped by the $Type$ field)
null	null	1	1	12	Grand total of 12 accounts (grpType = 1 and grpCty = 1 indicates this is the grand total)

You can use these query results to present a cross-tabular reports of the results.

Type/BillingCountry	USA	France	null	Total
Customer - Direct	1	0	6	7
Customer - Channel	1	1	2	4
null	1	0	0	1
Total	3	1	8	12

HAVING

HAVING is an optional clause that can be used in a SOQL query to filter results that aggregate functions return.

With API version 18.0 and later, you can use a HAVING clause with a GROUP BY clause to filter the results returned by aggregate functions, such as SUM(). The HAVING clause is similar to a WHERE clause. The difference is that you can include aggregate functions in a HAVING clause, but not in a WHERE clause. The syntax is:

```
[HAVING havingConditionExpression]
```

havingConditionExpression specifies one or more conditional expressions using aggregate functions to filter the query results

For example, you can use a GROUP BY clause to determine how many leads are associated with each LeadSource value with the following query:

```
SELECT LeadSource, COUNT(Name)
FROM Lead
GROUP BY LeadSource
```

However, if you are only interested in LeadSource values that have generated more than 100 leads, you can filter the results by using a HAVING clause. For example:

```
SELECT LeadSource, COUNT(Name)
FROM Lead
GROUP BY LeadSource
HAVING COUNT(Name) > 100
```

The next query returns accounts with duplicate names:

```
SELECT Name, Count(Id)
FROM Account
GROUP BY Name
HAVING Count(Id) > 1
```

For a list of aggregate functions supported by SOQL, see Aggregate Functions.

Considerations When Using HAVING

When you're creating SOQL queries with a HAVING clause, there are some considerations to keep in mind.

• A HAVING clause can filter by aggregated values. It can also contain filter by any fields included in the GROUP BY clause. To filter by any other field values, add the filtering condition to the WHERE clause. For example, the following query is valid:

```
SELECT LeadSource, COUNT(Name)
FROM Lead
GROUP BY LeadSource
HAVING COUNT(Name) > 100 and LeadSource > 'Phone'
```

The following guery is invalid as City is not included in the GROUP BY clause:

```
SELECT LeadSource, COUNT(Name)
FROM Lead
GROUP BY LeadSource
HAVING COUNT(Name) > 100 and City LIKE 'San%'
```

- Similar to a WHERE clause, a HAVING clause supports all the comparison operators, such as =, in conditional expressions, which can contain multiple conditions using the logical AND, OR, and NOT operators.
- A HAVING clause can't contain any semi- or anti-joins. A semi-join is a subquery on another object in an IN clause to restrict the records returned. An anti-join is a subquery on another object in a NOT IN clause to restrict the records returned.

TYPEOF

TYPEOF is an optional clause that can be used in a SELECT statement of a SOQL query when you're querying data that contains polymorphic relationships. A TYPEOF expression specifies a set of fields to select that depend on the runtime type of the polymorphic reference.

TYPEOF is available in API version 46.0 and later. (It is also available in earlier versions as part of a Developer Preview of the SOQL Polymorphism feature.)

```
SELECT [fieldList,]

[TYPEOF typeOfField

{WHEN whenObjectType THEN whenFieldList}[...]

[ELSE elseFieldList]

END][...]

FROM objectType
```

If you need to query multiple polymorphic relationship fields, you can use more than one TYPEOF expression in a single SELECT statement,

You can provide as many WHEN clauses as needed, one per object type. The ELSE clause is optional and used if the object type for the polymorphic relationship field in the current record doesn't match any of the object types in the provided WHEN clauses. The syntax specific to TYPEOF is as follows.

Syntax	Description
fieldList	Specifies a list of one or more fields, separated by commas, that you want to retrieve from the specified objectType . This is the standard list of fields used in a SELECT statement and used regardless of polymorphic relationship object types. If you're only interested in fields from objects referenced by polymorphic relationships, you can omit this list from your SELECT statement. This list of fields cannot reference relationship fields that are also referenced in any typeOfField fields used in the same query.
typeOfField	A polymorphic relationship field in objectType or a polymorphic field in a parent of objectType that can reference multiple object types. For example, the Who relationship field of a Task can be a Contact or a Lead. typeOfField cannot reference a relationship field that is also referenced in fieldList .
whenObjectType	An object type for the given WHEN clause. When the SELECT statement runs, each object type associated with the polymorphic relationship field specified in the <code>typeOfField</code> expression is checked for a matching object type in a WHEN clause.
whenFieldList	A list of one or more fields, separated by commas, that you want to retrieve from the specified whenObjectType . These are fields in the referenced object type or paths to related object fields, not fields in the primary object type for the SELECT statement.
elseFieldList	A list of one or more fields, separated by commas, that you want to retrieve if none of the WHEN clauses match the object type associated with the polymorphic relationship field specified in typeOfField . This list may only contain fields valid for the Name object type, or paths to related object fields in Name.
objectType	Specifies the type of object you want to query. This is the standard object type required in a SELECT statement.

Note the following considerations for using TYPEOF:

- TYPEOF can't be used with a relationship field whose namePointing attribute is false.
- TYPEOF is only allowed in the SELECT clause of a query. You can filter on the object type of a polymorphic relationship using the Type qualifier in a WHERE clause, see Filtering on Polymorphic Relationship Fields for more details.
- TYPEOF isn't allowed in queries that don't return objects, such as COUNT() and aggregate queries.
- TYPEOF can't be used in SOQL queries that are the basis of Streaming API PushTopics.
- TYPEOF can't be used in SOQL used in Bulk API.
- TYPEOF expressions can't be nested. For example, you can't use TYPEOF inside the WHEN clause of another TYPEOF expression.
- TYPEOF isn't allowed in the SELECT clause of a semi-join query. You can use TYPEOF in the SELECT clause of an outer query that contains semi-join queries. The following example is not valid:

```
SELECT Name FROM Account
WHERE CreatedById IN

(
SELECT
TYPEOF Owner
WHEN User THEN Id
WHEN Group THEN CreatedById
END
FROM CASE
)
```

The following semi-join clause is valid because TYPEOF is only used in the outer SELECT clause:

```
SELECT

TYPEOF What

WHEN Account THEN Phone
ELSE Name
END
FROM Event
WHERE CreatedById IN

(
SELECT CreatedById
FROM Case
)
```

GROUP BY, GROUP BY ROLLUP, GROUP BY CUBE, and HAVING aren't allowed in queries that use TYPEOF.

The following example selects specific fields depending on whether the What field of an Event references an Account or Opportunity.

```
SELECT

TYPEOF What

WHEN Account THEN Phone, NumberOfEmployees

WHEN Opportunity THEN Amount, CloseDate

ELSE Name, Email

END

FROM Event
```

See Understanding Relationship Fields and Polymorphic Fields for more details on polymorphic relationships, and more examples of TYPEOF.

FORMAT ()

Use FORMAT with the SELECT clause to apply localized formatting to standard and custom number, date, time, and currency fields.

When the FORMAT function is applied these fields reflect the appropriate format for the given user locale. The field format matches what appears in the Salesforce Classic user interface. For example, the date December 28, 2015 can appear numerically as 2015-12-28, 28-12-2015, 28/12/2015, 12/28/2015, or 28.12.2015, depending on the org's locale setting.

This example is for an org that has multiple currencies enabled.

```
SELECT FORMAT(amount) Amt, format(lastModifiedDate) editDate FROM Opportunity editDate = "7/2/2015 3:11 AM"

Amt = "AED 1,500.000000 (USD 1,000.00)"
```

The FORMAT function supports aliasing. In addition, aliasing is required when the query includes the same field multiple times. For example:

```
SELECT Id, LastModifiedDate, FORMAT(LastModifiedDate) formattedDate FROM Account
```

You can also nest it with aggregate or convertCurrency () functions. For example:

```
SELECT amount, FORMAT(amount) Amt, convertCurrency(amount) editDate,
FORMAT(convertCurrency(amount)) convertedCurrency FROM Opportunity where id = '12345'
SELECT FORMAT(MIN(closedate)) Amt FROM opportunity
```

FOR VIEW

Salesforce stores information about record views in the interface and uses the information to generate a list of recently viewed and referenced records, such as a list of records in a sidebar and for a list of records as auto-complete options in search. You can update objects with information about when they were last viewed by using the FOR VIEW clause in a SOQL query.

Consider using the FOR VIEW clause in conjunction with the FOR REFERENCE clause to update recent usage data for retrieved objects.

When this clause is used with a query, two things happen:

- The LastViewedDate field for the retrieved record is updated.
- A record is added to the RecentlyViewed object to reflect the recently viewed data for the retrieved record.



Note:

- Use this clause only when you are sure that the retrieved records will definitely be viewed by the logged-in user, else the clause falsely updates the usage information for the records. Also, the user won't recognize any falsely updated records when they display in the Recent Items and the global search auto-complete lists.
- The RecentlyViewed object is updated every time the logged-in user views or references a record. It is also updated when records are retrieved using the FOR VIEW or FOR REFERENCE clause in a SOQL query. To ensure that the most recent data is available, RecentlyViewed data is periodically truncated down to 200 records per object. RecentlyViewed data is retained for 90 days, after which it is removed on a periodic basis.

This is an example of a SOQL query that retrieves one contact to show to the current user and uses FOR VIEW to update the last viewed date of the retrieved contact. The same statement both retrieves the record and updates its last viewed date.

```
SELECT Name, ID FROM Contact LIMIT 1 FOR VIEW
```

FOR REFERENCE

FOR REFERENCE is an optional clause that can be used in a SOQL query to notify Salesforce when a record is referenced from a custom interface, such as in a mobile application or from a custom page.

A record is referenced every time a related record is viewed. For example, when a user views an account record, all related records (such as contacts, owner, leads, opportunities) are referenced. Consider using the FOR REFERENCE clause with the FOR VIEW clause to update recent usage data for retrieved objects.

When this clause is used with a query, two things happen:

- The LastReferencedDate field is updated for any retrieved records.
- A record is added to the RecentlyViewed object to reflect the recently referenced data for each retrieved record.



Note:

- Use this clause only when you're certain that the records affected by the query will be referenced, else the clause falsely updates the recently referenced information for any retrieved records. Also, the user won't recognize any falsely updated records when they display in the Recent Items and the global search auto-complete lists.
- The RecentlyViewed object is updated every time the logged-in user views or references a record. It is also updated when records are retrieved using the FOR VIEW or FOR REFERENCE clause in a SOQL query. To ensure that the most recent data is available, RecentlyViewed data is periodically truncated down to 200 records per object. RecentlyViewed data is retained for 90 days, after which it is removed on a periodic basis.

This is an example of a SOQL query that retrieves a contact and uses FOR REFERENCE to update the LastReferencedDate field of the retrieved contact.

SELECT Name, ID FROM Contact LIMIT 1 FOR REFERENCE

FOR UPDATE

In Apex, you can use FOR UPDATE to lock sObject records while they're being updated in order to prevent race conditions and other thread safety problems.

While an sObject record is locked, no other client or user is allowed to make updates either through code or the Salesforce user interface. The client locking the records can perform logic on the records and make updates with the guarantee that the locked records won't be changed by another client during the lock period. The lock gets released when the transaction completes.

To lock a set of sObject records in Apex, embed the keywords FOR UPDATE after any inline SOQL statement. For example, the following statement, in addition to guerying for two accounts, also locks the accounts that are returned:

Account [] accts = [SELECT Id FROM Account LIMIT 2 FOR UPDATE];



Note: You can't use the ORDER BY keywords in any SOQL query that uses locking.

Locking Considerations

- While the records are locked by a client, the locking client can modify their field values in the database in the same transaction. Other clients have to wait until the transaction completes and the records are no longer locked before being able to update the same records. Other clients can still query the same records while they're locked.
- If you attempt to lock a record currently locked by another client, your process waits for the lock to be released before acquiring a
 new lock. If the lock isn't released within 10 seconds, you will get a QueryException. Similarly, if you attempt to update a record
 currently locked by another client and the lock isn't released within 10 seconds, you will get a DmlException.

- If a client attempts to modify a locked record, the update operation might succeed if the lock gets released within a short amount of time after the update call was made. In this case, it is possible that the updates will overwrite those made by the locking client if the second client obtained an old copy of the record. To prevent this from happening, the second client must lock the record first. The locking process returns a fresh copy of the record from the database through the SELECT statement. The second client can use this copy to make new updates.
- When you perform a DML operation on one record, related records are locked in addition to the record in question. For more information, see the Record Locking Cheat Sheet.



Aggregate Functions

Use aggregate functions in a GROUP BY clause in SOQL queries to generate reports for analysis. Aggregate functions include AVG(), COUNT(), MIN(), MAX(), SUM(), and more.

Warning: Use care when setting locks in your Apex code. See Avoiding Deadlocks in the Apex Guide for more information.

You can also use aggregate functions without using a GROUP BY clause. For example, you could use the AVG () aggregate function to find the average Amount for all your opportunities.

```
SELECT AVG (Amount)
FROM Opportunity
```

However, these functions become a more powerful tool to generate reports when you use them with a GROUP BY clause. For example, you could find the average Amount for all your opportunities by campaign.

```
SELECT CampaignId, AVG(Amount)
FROM Opportunity
GROUP BY CampaignId
```

This table lists all the aggregate functions supported by SOQL.

Aggregate Function	Description						
AVG()	Returns the average value of a numeric field. For example:						
	SELECT CampaignId, AVG(Amount) FROM Opportunity GROUP BY CampaignId						
	Available in API version 18.0 and later.						
COUNT() and COUNT(fieldName)	Returns the number of rows matching the query criteria. For example using COUNT ():						
	SELECT COUNT() FROM Account WHERE Name LIKE 'a%'						
	For example using COUNT (fieldName):						
	SELECT COUNT(Id) FROM Account WHERE Name LIKE 'a%'						
	Note: COUNT (Id) in SOQL is equivalent to COUNT (*) in SQL.						

Aggregate Function	Description						
	The COUNT (fieldName) syntax is available in API version 18.0 and later. If you are using a GROUP BY clause, use COUNT (fieldName) instead of COUNT (). For more information, see COUNT () and COUNT (fieldName).						
COUNT_DISTINCT()	Returns the number of distinct non-null field values matching the query criteria. For example:						
	SELECT COUNT_DISTINCT(Company) FROM Lead						
	Note: COUNT_DISTINCT (fieldName) in SOQL is equivalent to COUNT (DISTINCT fieldName) in SQL. To query for all the distinct values, including null, for an object, see GROUP BY. Available in API version 18.0 and later.						
MIN()	Returns the minimum value of a field. For example:						
	SELECT MIN(CreatedDate), FirstName, LastName FROM Contact GROUP BY FirstName, LastName						
	If you use the MIN () or MAX () functions on a picklist field, the function uses the sort order of the picklist values instead of alphabetical order.						
	Available in API version 18.0 and later.						
MAX()	Returns the maximum value of a field. For example:						
	SELECT Name, MAX(BudgetedCost) FROM Campaign GROUP BY Name						
	Available in API version 18.0 and later.						
SUM()	Returns the total sum of a numeric field. For example:						
	SELECT SUM(Amount) FROM Opportunity WHERE IsClosed = false AND Probability > 60						
	Available in API version 18.0 and later.						

You can't use a LIMIT clause in a query that uses an aggregate function, but does not use a GROUP BY clause. For example, the following query is invalid:

```
SELECT MAX(CreatedDate)
FROM Account LIMIT 1
```

COUNT() and COUNT(fieldName)

COUNT () is an optional clause that can be used in a SELECT statement in a SOQL query to discover the number of rows that a query returns.

There are two versions of syntax for COUNT ():

- COUNT()
- COUNT (fieldName)

If you are using a GROUP BY clause, use COUNT (fieldName) instead of COUNT ().

COUNT()

COUNT () returns the number of rows that match the filtering conditions.

For example:

```
SELECT COUNT()
FROM Account
WHERE Name LIKE 'a%'
SELECT COUNT()
FROM Contact, Contact. Account
WHERE Account.Name = 'MyriadPubs'
```

For COUNT(), the query result size field returns the number of rows. The records field returns null.

Note the following when using COUNT ():

- COUNT () must be the only element in the SELECT list.
- You can use COUNT () with a LIMIT clause.
- You can't use COUNT() with an ORDER BY clause. Use COUNT(fieldName) instead.
- You can't use COUNT () with a GROUP BY clause for API version 19.0 and later. Use COUNT (fieldName) instead.

COUNT (fieldName)

COUNT (fieldName) returns the number of rows that match the filtering conditions and have a non-null value for fieldName. This syntax is newer than COUNT () and is available in API version 18.0 and later.

For example:

```
SELECT COUNT(Id)
FROM Account
WHERE Name LIKE 'a%'
```

COUNT (Id) returns the same count as COUNT (), so the previous and next queries are equivalent:

```
SELECT COUNT()
FROM Account
WHERE Name LIKE 'a%'
```



Note: COUNT (Id) in SOQL is equivalent to COUNT (*) in SQL.

For COUNT (fieldName), the AggregateResult object in the records field returns the number of rows. The size field does not reflect the count. For example:

```
SELECT COUNT(Id)
FROM Account
WHERE Name LIKE 'a%'
```

For this query, the count is returned in the expr0 field of the AggregateResult object. For more information, see Using Aliases with GROUP BY.

There are advantages to using COUNT (**fieldName**) instead of COUNT (). You can include multiple COUNT (**fieldName**) items in a SELECT clause. For example, the following query returns the number of opportunities, as well as the number of opportunities associated with a campaign.

```
SELECT COUNT(Id), COUNT(CampaignId)
FROM Opportunity
```

Unlike COUNT (), you can use a GROUP BY clause with COUNT (**fieldName**) in API version 18.0 and later. This allows you to analyze your records and return summary reporting information. For example, the following query returns the number of leads for each LeadSource value:

```
SELECT LeadSource, COUNT(Name)
FROM Lead
GROUP BY LeadSource
```

Support for Field Types in Aggregate Functions

Using aggregate functions in SOQL queries is a powerful way to analyze records, but the functions aren't relevant for all field types. For example, base64 fields don't support aggregate functions, because they wouldn't generate any meaningful data.

Aggregate functions are supported for several primitive data types and field types. The following table lists support by the aggregate functions for the primitive data types.

Data Type	AVG()	COUNT()	COUNT_DISTINCT()	MIN()	MAX()	SUM()
base64	No	No	No	No	No	No
boolean	No	No	No	No	No	No
byte	No	No	No	No	No	No
date	No	Yes	Yes	Yes	Yes	No
dateTime	No	Yes	Yes	Yes	Yes	No
double	Yes	Yes	Yes	Yes	Yes	Yes
int	Yes	Yes	Yes	Yes	Yes	Yes
string	No	Yes	Yes	Yes	Yes	No
time	No	No	No	No	No	No

In addition to the primitive data types, the API uses an extended set of field types for object fields. The following table lists support by the aggregate functions for these field types.

Data Type	AVG()	COUNT()	COUNT_DISTINCT()	MIN()	MAX()	SUM()
address	No	No	No	No	No	No
anyType	No	No	No	No	No	No

Data Type	AVG()	COUNT()	COUNT_DISTINCT()	MIN()	MAX()	SUM()
calculated	Depends on data type*	Depends on data type*	Depends on data type	Depends on data type*	Depends on data type*	Depends on data type*
combobox	No	Yes	Yes	Yes	Yes	No
currency**	Yes	Yes	Yes	Yes	Yes	Yes
DataCategoryGroupReference	No	Yes	Yes	Yes	Yes	No
email	No	Yes	Yes	Yes	Yes	No
encryptedstring	No	No	No	No	No	No
location	No	No	No	No	No	No
ID	No	Yes	Yes	Yes	Yes	No
masterrecord	No	Yes	Yes	Yes	Yes	No
multipicklist	No	No	No	No	No	No
percent	Yes	Yes	Yes	Yes	Yes	Yes
phone	No	Yes	Yes	Yes	Yes	No
picklist	No	Yes	Yes	Yes	Yes	No
reference	No	Yes	Yes	Yes	Yes	No
textarea	No	Yes	Yes	Yes	Yes	No
url	No	Yes	Yes	Yes	Yes	No

 $^{^*}$ Calculated fields are custom fields defined by a formula, which is an algorithm that derives its value from other fields, expressions, or values. Therefore, support for aggregate functions depends on the type of the calculated field.

^{**} Aggregate function results on currency fields default to the system currency.



🁔 Tip: Some object fields have a field type that does not support grouping. You can't include fields with these field types in a GROUP BY clause. The Field object associated with DescribeSObjectResult has a groupable field that defines whether you can include the field in a GROUP BY clause.

Date Functions

Date functions in SOQL queries allow you to group or filter data by date periods such as day, calendar month, or fiscal year.

For example, you could use the CALENDAR YEAR () function to find the sum of the Amount values for all your opportunities for each calendar year.

SELECT CALENDAR YEAR (CreatedDate), SUM (Amount) FROM Opportunity GROUP BY CALENDAR YEAR (CreatedDate)

Date functions are available in API version 18.0 and later.



Note: SOQL queries in a client application return dateTime field values as Coordinated Universal Time (UTC) values. To convert dateTime field values to your default time zone, see Converting Time Zones in Date Functions.

This table lists all the date functions supported by SOQL.

Date Function	Description	Examples
CALENDAR_MONTH()	Returns a number representing the calendar month of a date field.	1 for January12 for December
CALENDAR_QUARTER()	Returns a number representing the calendar quarter of a date field.	 1 for January 1 through March 31 2 for April 1 through June 30 3 for July 1 through September 30 4 for October 1 through December 31
CALENDAR_YEAR()	Returns a number representing the calendar year of a date field.	2009
DAY_IN_MONTH()	Returns a number representing the day in the month of a date field.	20 for February 20
DAY_IN_WEEK()	Returns a number representing the day of the week for a date field.	1 for Sunday7 for Saturday
DAY_IN_YEAR()	Returns a number representing the day in the year for a date field.	32 for February 1
DAY_ONLY()	Returns a date representing the day portion of a dateTime field.	2009-09-22 for September 22, 2009 You can only use DAY_ONLY () with dateTime fields.
FISCAL_MONTH()	Returns a number representing the fiscal month of a date field. This differs from CALENDAR_MONTH() if your organization uses a fiscal year that does not match the Gregorian calendar. Note: This function is not supported if your organization has custom fiscal years enabled. See "Define Your Fiscal Year" in the Salesforce Help.	If your fiscal year starts in March: 1 for March 12 for February See "Set the Fiscal Year" in the Salesforce online help.
FISCAL_QUARTER()	Returns a number representing the fiscal quarter of a date field. This differs from CALENDAR_QUARTER () if your organization uses a fiscal year that does not match the Gregorian calendar. Note: This function is not supported if your organization has custom fiscal years enabled. See "Define Your Fiscal Year" in the Salesforce Help.	If your fiscal year starts in July: 1 for July 15 4 for June 6

Date Function	Description	Examples
FISCAL_YEAR()	Returns a number representing the fiscal year of a date field. This differs from CALENDAR_YEAR() if your organization uses a fiscal year that does not match the Gregorian calendar.	2009
	Note: This function is not supported if your organization has custom fiscal years enabled. See "Define Your Fiscal Year" in the Salesforce Help.	
HOUR_IN_DAY()	Returns a number representing the hour in the day for a	18 for a time of 18:23:10
	dateTime field.	You can only use <code>HOUR_IN_DAY()</code> with dateTime fields.
WEEK_IN_MONTH()	Returns a number representing the week in the month	2 for April 10
	for a date field.	The first week is from the first through the seventh day of the month.
WEEK_IN_YEAR()	Returns a number representing the week in the year for	1 for January 3
	a date field.	The first week is from January 1 through January 7.

Note the following when you use date functions:

• You can use a date function in a WHERE clause to filter your results even if your query doesn't include a GROUP BY clause. The following query returns data for 2009:

```
SELECT CreatedDate, Amount
FROM Opportunity
WHERE CALENDAR_YEAR(CreatedDate) = 2009
```

• You can't compare the result of a date function with a date literal in a WHERE clause. The following query doesn't work:

```
SELECT CreatedDate, Amount
FROM Opportunity
WHERE CALENDAR_YEAR(CreatedDate) = THIS_YEAR
```

• You can't use a date function in a SELECT clause unless you also include it in the GROUP BY clause. There is an exception if the field used in the date function is a date field. You can use the date field instead of the date function in the GROUP BY clause. This doesn't work for dateTime fields. The following query doesn't work because CALENDAR_YEAR (CreatedDate) is not in a GROUP BY clause:

```
SELECT CALENDAR_YEAR(CreatedDate), Amount FROM Opportunity
```

The following query works because the date field, CloseDate, is in the GROUP BY clause. This wouldn't work for a dateTime field, such as CreatedDate.

```
SELECT CALENDAR_YEAR(CloseDate)
FROM Opportunity
GROUP BY CALENDAR_YEAR(CloseDate)
```

Converting Time Zones in Date Functions

SOQL queries in a client application return dateTime field values as Coordinated Universal Time (UTC) values. You can use convertTimezone() in a date function to convert dateTime fields to the user's time zone.

For example, you could use the convertTimezone (**dateTimeField**) function to find the sum of the Amount values for all your opportunities for each hour of the day, where the hour is converted to the user's time zone.

```
SELECT HOUR_IN_DAY(convertTimezone(CreatedDate)), SUM(Amount)
FROM Opportunity
GROUP BY HOUR_IN_DAY(convertTimezone(CreatedDate))
```

Note that you can only use convertTimezone () in a date function. The following query doesn't work because there is no date function.

```
SELECT convertTimezone(CreatedDate)
FROM Opportunity
```

Querying Currency Fields in Multi-Currency Orgs

Use convertCurrency () in the SELECT statement of a SOQL query to convert currency fields to the user's currency. This action requires that the org has multiple currencies enabled.

The following syntax is for using convertCurrency() with the SELECT clause:

```
convertCurrency(field)
```

For example:

```
SELECT Id, convertCurrency(AnnualRevenue)
FROM Account
```

Use an ISO code that your org has enabled and made active. If you don't put in an ISO code, the numeric value is used instead of comparative amounts. Using the previous example, opportunity records with JPY5001, EUR5001, and USD5001 would be returned. If you use IN in a WHERE clause, you can't mix ISO code and non-ISO code values.

To format currencies according to the user's local, use FORMAT() with SELECT() statements. In this example, convertedCurrency is an alias for the returned amount, which is formatted appropriately in the user interface.

```
SELECT Amount, FORMAT(amount) Amt, convertCurrency(amount) convertedAmount, FORMAT(convertCurrency(amount)) convertedCurrency FROM Opportunity where id = '006R00000024gDtiAI'
```

If an org has enabled advanced currency management, dated exchange rates are used when converting currency fields on opportunities, opportunity line items, and opportunity history. With advanced currency management, convertCurrency uses the conversion rate that corresponds to a given field (for example, CloseDate on opportunities). When advanced currency management isn't enabled, the most recent conversion date entered is used.

Considerations and Workgrounds

You can't use the convertCurrency () function in a WHERE clause. If you do, an error is returned. Use the following syntax to convert a numeric value to the user's currency from any active currency in your org.

```
WHERE Object_name Operator ISO_CODEvalue
```

For example:

```
SELECT Id, Name
FROM Opportunity
WHERE Amount > USD5000
```

In this example, opportunity records are returned if the record's currency Amount value is greater than the equivalent of USD5000. For example, an opportunity with an amount of USD5001 is returned, but not JPY7000.

You can't convert the result of an aggregate function into the user's currency by calling the convertCurrency() function. If a query includes a GROUP BY or HAVING clause, currency data returned by using an aggregate function, such as SUM() or MAX(), is in the org's default currency.

For example:

```
SELECT Name, MAX(Amount)
FROM Opportunity
GROUP BY Name
HAVING MAX(Amount) > 10000
```

You can't use ISO_CODEvalue to represent a value in a particular currency, such as USD, when you use an aggregate function. For example, the following query doesn't work.

```
SELECT Name, MAX(Amount)

FROM Opportunity

GROUP BY Name

HAVING MAX(Amount) > USD10000
```

You can't use convertCurrency () with ORDER BY. Ordering is always based on the converted currency value, just like in reports.

Example SELECT Clauses

The following are examples of text searches that use SOQL.

Type of Search	Example(s)
Simple query	SELECT Id, Name, BillingCity FROM Account
WHERE	SELECT Id FROM Contact WHERE Name LIKE 'A%' AND MailingCity = 'California'
ORDER BY	SELECT Name FROM Account ORDER BY Name DESC NULLS LAST
LIMIT	SELECT Name FROM Account WHERE Industry = 'media' LIMIT 125
ORDER BY With LIMIT	SELECT Name FROM Account WHERE Industry = 'media' ORDER BY BillingPostalCode ASC NULLS LAST LIMIT 125
count()	SELECT COUNT() FROM Contact

Type of Search	Example(s)
GROUP BY	SELECT LeadSource, COUNT(Name) FROM Lead GROUP BY LeadSource
HAVING	SELECT Name, COUNT(Id) FROM Account GROUP BY Name HAVING COUNT(Id) > 1
OFFSET with ORDER BY	SELECT Name, Id FROM Merchandisec ORDER BY Name OFFSET 100
OFFSET with ORDER BY and LIMIT	SELECT Name, Id FROM Merchandisec ORDER BY Name LIMIT 20 OFFSET 100
Relationship queries: child-to-parent	SELECT Contact.FirstName, Contact.Account.Name FROM Contact
	SELECT Id, Name, Account.Name FROM Contact WHERE Account.Industry = 'media'
Relationship queries: parent-to-child	SELECT Name, (SELECT LastName FROM Contacts) FROM Account
	SELECT Account.Name, (SELECT Contact.LastName FROM Account.Contacts) FROM Account
Relationship query with WHERE	<pre>SELECT Name, (SELECT LastName FROM Contacts WHERE CreatedBy.Alias = 'x') FROM Account WHERE Industry = 'media'</pre>
Relationship query: child-to parent with custom objects	SELECT Id, FirstName_c, Mother_of_Child_r.FirstName_c FROM Daughter_c WHERE Mother_of_Child_r.LastName_c LIKE 'C%'
Relationship query: parent to child with custom objects	SELECT Name, (SELECT Name FROM Line_Itemsr) FROM Merchandisec WHERE Name LIKE 'Acme%'
Relationship queries with polymorphic key	SELECT Id, Owner.Name FROM Task WHERE Owner.FirstName like 'B%'
	SELECT Id, Who.FirstName, Who.LastName FROM Task WHERE Owner.FirstName LIKE 'B%'
	SELECT Id, What.Name FROM Event
Polymorphic relationship queries using TYPEOF	SELECT TYPEOF What WHEN Account THEN Phone, NumberOfEmployees WHEN Opportunity THEN Amount, CloseDate ELSE Name, Email END FROM Event
Relationship queries with aggregate	SELECT Name, (SELECT CreatedBy.Name FROM Notes) FROM Account
	SELECT Amount, Id, Name, (SELECT Quantity, ListPrice, PricebookEntry.UnitPrice, PricebookEntry.Name FROM OpportunityLineItems) FROM Opportunity

Type of Search	Example(s)
Simple query: the Userld and LoginTime for each user	SELECT UserId, LoginTime from LoginHistory
Relationship queries with number of logins per user in a specific time range	SELECT UserId, COUNT(Id) from LoginHistory WHERE LoginTime > 2010-09-20T22:16:30.000Z AND LoginTime < 2010-09-21T22:16:30.000Z GROUP BY UserId
Query to retrieve the RecordType per user	SELECT Id, Name, IsActive, SobjectType, DeveloperName, Description FROM RecordType
	Note: If a user doesn't have Read access to the object to which a record type belongs, that record type isn't returned in the query results.



Note: Apex requires that you surround SOQL and SOSL statements with square brackets to use them on the fly. You can use Apex script variables and expressions when preceded by a colon (:).

Relationship Queries

Client applications need to be able to query for more than a single type of object at a time. SOQL provides syntax to support these types of queries, called *relationship queries*, against standard objects and custom objects. Relationship queries traverse parent-to-child and child-to-parent relationships between objects to filter and return results.

Relationship queries are similar to SQL joins. However, you cannot perform arbitrary SQL joins. The relationship queries in SQQL must traverse a valid relationship path as defined in the rest of this section.

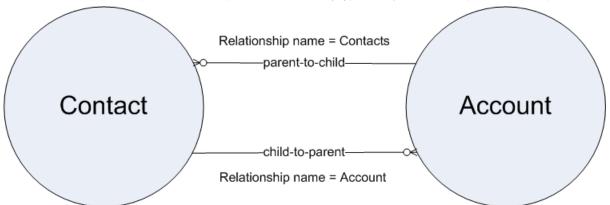
You can use relationship queries to return objects of one type based on criteria that applies to objects of another type, for example, "return all accounts created by Bob Jones and the contacts associated with those accounts." There must be a parent-to-child or child-to-parent relationship connecting the objects. You can't write arbitrary queries such as "return all accounts and users created by Bob Jones."

Use the following topics to understand and use relationship queries in SOQL.

- Understanding Relationship Names
- Using Relationship Queries
- Understanding Relationship Names, Custom Objects, and Custom Fields
- Understanding Query Results
- Lookup Relationships and Outer Joins
- Identifying Parent and Child Relationships
- Understanding Relationship Fields and Polymorphic Fields
- Understanding Relationship Query Limitations
- Using Relationship Queries with History Objects
- Using Relationship Queries with Data Category Selection Objects
- Using Relationship Queries with the Partner WSDL

Understanding Relationship Names

Parent-to-child and child-to-parent relationships exist between many types of objects. For example, Account is a parent of Contact.



To be able to traverse these relationships for standard objects, a relationship name is given to each relationship. The form of the name is different, depending on the direction of the relationship:

• For child-to-parent relationships, the relationship name to the parent is the name of the foreign key, and there is a relationshipName property that holds the reference to the parent object. For example, the Contact child object has a child-to-parent relationship to the Account object, so the value of relationshipName in Contact is Account. These relationships are traversed by specifying the parent using dot notation in the guery, for example:

```
SELECT Contact.FirstName, Contact.Account.Name from Contact
```

This query returns the first names of all the contacts in the organization, and for each contact, the account name associated with (parent of) that contact.

• For parent-to-child relationships, the parent object has a name for the child relationship that is unique to the parent, the plural of the child object name. For example, Account has child relationships to Assets, Cases, and Contacts among other objects, and has a relationshipName for each, Assets, Cases, and Contacts. These relationships can be traversed only in the SELECT clause, using a nested SOQL query. For example:

SELECT Account.Name, (SELECT Contact.FirstName, Contact.LastName FROM Account.Contacts) FROM Account

This query returns all accounts, and for each account, the first and last name of each contact associated with (the child of) that account.

Warning: You must use the correct naming convention and SELECT syntax for the direction of the relationship. For information about how to discover relationship names via your organization's WSDL or describeSObjects(), see Identifying Parent and Child Relationships. There are limitations on relationship queries depending on the direction of the relationship. See Understanding Relationship Query Limitations for more information.

Relationship names are somewhat different for custom objects, though the SELECT syntax is the same. See Identifying Parent and Child Relationships for more information.

Using Relationship Queries

Use SOQL to query several relationship types.

You can query the following relationships using SOQL:

• Query child-to-parent relationships, which are often many-to-one. Specify these relationships directly in the SELECT, FROM, or WHERE clauses using the dot (.) operator.

For example:

```
SELECT Id, Name, Account.Name
FROM Contact
WHERE Account.Industry = 'media'
```

This query returns the ID and name for only the contacts whose related account industry is media, and for each contact returned, the account name.

• Query parent-to-child, which are almost always one-to-many. Specify these relationships using a subquery (enclosed in parentheses), where the initial member of the FROM clause in the subquery is related to the initial member of the outer query FROM clause. Note that for standard object subqueries, you should specify the plural name of the object as that is the name of the relationship for each object.

For example:

```
SELECT Name,
(
    SELECT LastName
    FROM Contacts
)
FROM Account
```

The query returns the name for all the accounts, and for each account, the last name of each contact.

Traverse the parent-to-child relationship as a foreign key in an aggregate query:

For example:

```
SELECT Name,

(
SELECT CreatedBy.Name
FROM Notes
)
FROM Account
```

This query returns the accounts in an organization, and for each account, the name of the account, the notes for those accounts (which can be an empty result set if there were no notes on any accounts) with the name of the user who created each note (if the result set is not empty).

• In a similar example, traverse the parent-to-child relationship in an aggregate guery:

```
SELECT Amount, Id, Name,

(
SELECT Quantity, ListPrice,
PricebookEntry.UnitPrice, PricebookEntry.Name
FROM OpportunityLineItems
)
FROM Opportunity
```

Using the same query, you can get the values on Product2 by specifying the product family (which points to the field's data):

```
SELECT Amount, Id, Name, (SELECT Quantity, ListPrice, PriceBookEntry.UnitPrice, PricebookEntry.Name,
```

```
PricebookEntry.product2.Family FROM OpportunityLineItems)
FROM Opportunity
```

• Any query (including subqueries) can include a WHERE clause, which applies to the object in the FROM clause of the current query.

These clauses can filter on any object in the current scope (reachable from the root element of the query), via the parent relationships.

For example:

```
SELECT Name,

(
SELECT LastName
FROM Contacts
WHERE CreatedBy.Alias = 'x')
FROM Account WHERE Industry = 'media'
```

This query returns the name for all accounts whose industry is media, and for each account returned, returns the last name of every contact whose created-by alias is 'x.'

Understanding Relationship Names, Custom Objects, and Custom Fields

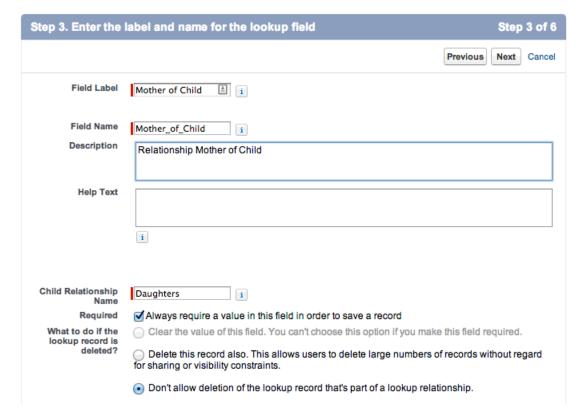
Custom objects can participate in relationship queries. Salesforce ensures that your custom object names, custom field names, and the relationship names that are associated with them remain unique, even if a standard object with the same name is available now or in the future. Having unique relationship queries is important in cases where the query traverses relationships that use the object, field, and relationship names.

This topic explains how relationship names for custom objects and custom fields are created and used.

When you create a new custom relationship in the Salesforce user interface, you are asked to specify the plural version of the object name, which you use for relationship queries:

New Relationship





Notice that the Child Relationship Name (parent to child) is the plural form of the child object name, in this case Daughters.

Once the relationship is created, notice that it has an API Name, which is the name of the custom field you created, appended by c (underscore-underscore-c):



When you refer to this field via the API, you must use this special form of the name. This prevents ambiguity in the case where Salesforce can create a standard object with the same name as your custom field. The same process applies to custom objects—when they are created, they have an API Name, the object named appended by c, which must be used.

When you use a relationship name in a query, you must use the relationship names without the $__c$. Instead, append an $__r$ (underscore underscore r).

For example:

• When you use a child-to-parent relationship, you can use dot notation:

```
SELECT Id, FirstName__c, Mother_of_Child__r.FirstName__c
FROM Daughter__c
WHERE Mother_of_Child__r.LastName__c LIKE 'C%'
```

This query returns the ID and first name of daughter objects, and the first name of the daughter's mother if the mother's last name begins with 'C.'

Parent-to-child relationship queries do not use dot notation:

```
SELECT LastName__c,

(
    SELECT LastName__c
    FROM Daughters__r
)
FROM Mother__c
```

The example above returns the last name of all mothers, and for each mother returned, the last name of the mother's daughters.

Understanding Query Results

Query results are returned as nested objects. The primary or "driving" object of the main SELECT statement in a SOQL query contains query results of subqueries.

For example, you can construct a guery using either parent-to-child or child-to-parent syntax:

• Child-to-parent:

```
SELECT Id, FirstName, LastName, AccountId, Account.Name
FROM Contact
WHERE Account.Name LIKE 'Acme%'
```

This query returns one query result (assuming there were not too many returned records), with a row for every contact that met the WHERE clause criteria.

Parent-to-child:

```
SELECT Id, Name,

(
SELECT Id, FirstName, LastName
FROM Contacts
)

FROM Account
WHERE Name like 'Acme%'
```

This query returns a set of accounts, and within each account, a query result set of Contact fields containing the contact information from the subquery.

Subquery results are like regular query results in that you might need to use queryMore () to retrieve all the records if there are many children. For example, if you issue a query on accounts that includes a subquery, your client application must handle results from the subquery as well:

- 1. Perform the query on Account.
- 2. Iterate over the account QueryResult with queryMore ().
- **3.** For each account object, retrieve the contacts QueryResult.
- **4.** Iterate over the child contacts, using queryMore () on each contact's QueryResult.

The following sample illustrates how to process subquery results:

```
private void querySample() {
 QueryResult qr = null;
 try {
   gr = connection.guery("SELECT a.Id, a.Name, " +
      "(SELECT c.Id, c.FirstName, " +
      "c.LastName FROM a.Contacts c) FROM Account a");
   boolean done = false;
   if (gr.getSize() > 0) {
     while (!done) {
       for (int i = 0; i < qr.getRecords().length; i++) {</pre>
       Account acct = (Account) qr.getRecords()[i];
       String name = acct.getName();
       System.out.println("Account " + (i + 1) + ": " + name);
       printContacts(acct.getContacts());
       if (qr.isDone()) {
        done = true;
       } else {
         qr = connection.queryMore(qr.getQueryLocator());
      }
    } else {
      System.out.println("No records found.");
   System.out.println("\nQuery successfully executed.");
 } catch (ConnectionException ce) {
   System.out.println("\nFailed to execute query successfully, error message " +
    "was: \n" + ce.getMessage());
 }
}
private void printContacts(QueryResult qr) throws ConnectionException {
 boolean done = false;
 if (gr.getSize() > 0) {
   while (!done) {
   for (int i = 0; i < qr.getRecords().length; i++) {</pre>
     Contact contact = (Contact) qr.getRecords()[i];
     String fName = contact.getFirstName();
     String lName = contact.getLastName();
     System.out.println("Child contact " + (i + 1) + ": " + lName
      + ", " + fName);
   if (qr.isDone()) {
     done = true;
    } else {
     gr = connection.queryMore(gr.getQueryLocator());
```

```
}
} else {
   System.out.println("No child contacts found.");
}
```

Lookup Relationships and Outer Joins

Beginning with API version 13.0, relationship SOQL queries return records, even if the relevant foreign key field has a null value, as with an outer join.

The change in behavior applies to the following types of relationship queries.

• In an ORDER BY clause, if the foreign key value in a record is null, the record is returned in version 13.0 and later, but not returned in versions before 13.0. For example:

```
SELECT Id, CaseNumber, Account.Id, Account.Name
FROM Case
ORDER BY Account.Name
```

Any case record for which AccountId is empty is returned in version 13.0 and later.

The following example uses custom objects:

```
SELECT ID, Name, Parent__r.id, Parent__r.name
FROM Child__c
ORDER BY Parent__r.name
```

This query returns the Id and Name values of the Child object and the Id and name of the Parent object referenced in each Child, and orders the results by the parent name. In version 13.0 and later, records are returned even if Parent__r.id or Parent__r.name are null. In earlier versions, such records are not returned by the query.

• In a WHERE clause that uses OR, if the foreign key value in a record is null, the record is returned in version 13.0 and later, but not returned in versions before 13.0. For example, if your organization has one contact with the value of its LastName field equal to foo and the value of its AccountId field equal to null, and another contact with a different last name and a parent account named bar, the following query returns only the contact with the last name equal to bar:

```
SELECT Id FROM Contact WHERE LastName = 'foo' or Account.Name = 'bar'
```

The contact with no parent account has a last name that meets the criteria, so it is returned in version 13.0 and later.

• In a WHERE clause that checks for a value in a parent field, if the parent does not exist, the record is returned in version 13.0 and later but is not returned in versions before 13.0. For example:

```
SELECT Id
FROM Case
WHERE Contact.LastName = null
```

Case record Id values are returned in version 13.0 and later, but are not returned in versions before 13.0.

• In a WHERE clause that uses a Boolean field, the Boolean field never has a null value. Instead, null is treated as false. Boolean fields on outer-joined objects are treated as false when no records match the guery.

Identifying Parent and Child Relationships

Identify parent-child relationships by viewing Entity Relationship Diagrams (ERD) or by examining the enterprise WSDL for your organization.

You can identify parent-child relationships by viewing the ERD diagrams in the Data Model section of the Salesforce Object Reference at www.salesforce.com/us/developer/docs/object_reference/index.htm. However, not all parent-child relationships are exposed in SOQL, so to be sure you can query on a parent-child relationship by issuing the appropriate describe call. The results contain parent-child relationship information.

You can also examine the enterprise WSDL for your organization:

• To find the names of child relationships, look for entries that contain the plural form of a child object and end with type="tns:QueryResult". For example, from Account:

In the example above, the child relationship name Contacts is in the entry for its parent Account.

• For the parent of an object, look for a pair of entries, such as AccountId and Account, where the ID field represents the parent object referenced by the ID, and the other represents the contents of the record. The parent entry has a non-primitive type, type="ens:Account".

- Note: Not all relationships are exposed in the API. The most reliable method for identifying relationships is to execute a describeSObjects() call. You can use the AJAX Toolkit to quickly execute test calls.
- For custom objects, look for a pair of entries with the relationship suffix r:

Understanding Relationship Fields and Polymorphic Fields

In a polymorphic relationship, the referenced object of the relationship can be one of several different types of object.

Some fields are *relationship* fields, which means they can be used to get information about a related object. And some of those relationship fields are *polymorphic* fields. A polymorphic field is one where the related object might be one of several different types of objects. For example, the Who relationship field of a Task can be a Contact or a Lead.

To determine what kind a field is, call describeSObjects() on the object and examine the properties for the field.

- 1. If relationshipName is not null, the field is a relationship field.
- 2. If, in addition, namePointing is true, then the field is polymorphic.

Example of a Relationship Field

The OwnerId field of the Account object has the following properties:

- relationshipName = Owner
- namePointing = false
- referenceTo = User

This means it is a relationship field. Use the relationshipName as a pseudo field to get information about the related object, whose type is specified in the referenceTo field. For example, you can use the following SOQL query:

```
SELECT Id, Owner.Name FROM CAMPAIGN
```



Note: Use the name of the relationship (Owner) not the name of the field (OwnerId).

Example of a Polymorphic Field

The OwnerId field of the Event object has the following properties:

- relationshipName = Owner
- namePointing = true
- referenceTo = Calendar, User

This means it is a polymorphic field. Owner could be a Calendar, or a User. For example, you can use the following SOQL query:

```
SELECT Id, Owner.Name FROM Event WHERE Owner.Type = 'User'
```

Using Polymorphic Fields

You can use polymorphic fields several ways.

- You can use the polymorphic field for the relationship.
- You can use the Type qualifier on a polymorphic field.
- You can use a TYPEOF clause in a query.

You can also combine these techniques for complex queries.

Common Polymorphic Fields

Common polymorphic fields include:

• Owner: This field represents the parent of the record. For example:

```
SELECT Id, Owner.Name
FROM Task
WHERE Owner.FirstName like 'B%'
```

This example query works for Task records whose owners are either Calendars or Users.

• Who: This field represents the person associated with the record. For example:

```
SELECT Id, Who.FirstName, Who.LastName
FROM Task
WHERE Owner.FirstName LIKE 'B%'
```

This example query works for Task records whose owners can be either Calendars or Users, and whose who field can be either Contacts or Leads.

If you'd like to know the type of object returned in a query, use Who. Type. For example:

```
SELECT Id, Who.Id, Who.Type
FROM Task
```

Using this example, you can query all the Tasks associated with Contacts:

```
SELECT Id, Who.Id, Who.Type
FROM Task
WHERE Who.Type='Contact'
```

What: This field represents nonhuman objects that are associated with the record. For example:

```
SELECT Id, What.Name
FROM Event
```

This example query works for Events where What can be an Account or a Solution, or any of another number of object types.

Using the Type Qualifier

You can use the Type qualifier on a field to determine the object type that's referenced in a polymorphic relationship. Use the Type qualifier in the WHERE clause of a SELECT statement to conditionally control what's returned from the query depending on the referenced object type. The following SELECT statement uses Type to filter the query based on the What field in Event.

```
SELECT Id
FROM Event
WHERE What.Type IN ('Account', 'Opportunity')
```

At run time, this SELECT statement returns the IDs for Events that reference Accounts or Opportunities in the What field. If an Event referenced a Campaign in the What field, it wouldn't be returned as part of this SELECT. Unlike the TYPEOF expression, object types are returned as strings from Type. You can apply any WHERE comparison operator to the object type strings, such as = (Equals) or LIKE.

Using TYPEOF

SOQL supports polymorphic relationships using the TYPEOF expression in a SELECT statement. TYPEOF is available in API version 46.0 and later. (It is also available in API version 26.0 and later as part of a Developer Preview.)

Use TYPEOF in a SELECT statement to control which fields to query for each object type in a polymorphic relationship. The following SELECT statement returns a different set of fields depending on the object type associated with the What polymorphic relationship field in an Event.

```
SELECT

TYPEOF What

WHEN Account THEN Phone, NumberOfEmployees

WHEN Opportunity THEN Amount, CloseDate

ELSE Name, Email

END

FROM Event
```

At run time, this SELECT statement checks the object type referenced by the What field in an Event. If the object type is Account, the referenced Account's Phone and NumberOfEmployee fields are returned. If the object type is Opportunity, the referenced Opportunity's Amount and CloseDate fields are returned. If the object type is any other type, the Name and Email fields are returned. Note that if an ELSE clause isn't provided and the object type isn't Account or Opportunity, then null is returned for that Event.

Note the following considerations for TYPEOF.

- TYPEOF can't be used with a relationship field whose namePointing attribute is false.
- TYPEOF is only allowed in the SELECT clause of a query. You can filter on the object type of a polymorphic relationship using the Type qualifier in a WHERE clause, see Filtering on Polymorphic Relationship Fields for more details.
- TYPEOF isn't allowed in queries that don't return objects, such as COUNT() and aggregate queries.
- TYPEOF can't be used in SOQL queries that are the basis of Streaming API PushTopics.
- TYPEOF can't be used in SOOL used in Bulk API.
- TYPEOF expressions can't be nested. For example, you can't use TYPEOF inside the WHEN clause of another TYPEOF expression.

• TYPEOF isn't allowed in the SELECT clause of a semi-join query. You can use TYPEOF in the SELECT clause of an outer query that contains semi-join queries. The following example is not valid:

```
SELECT Name FROM Account
WHERE CreatedById IN

(
SELECT
TYPEOF Owner
WHEN User THEN Id
WHEN Group THEN CreatedById
END
FROM CASE
)
```

The following semi-join clause is valid because TYPEOF is only used in the outer SELECT clause:

```
SELECT

TYPEOF What

WHEN Account THEN Phone
ELSE Name
END
FROM Event
WHERE CreatedById IN

(
SELECT CreatedById
FROM Case
)
```

• GROUP BY, GROUP BY ROLLUP, GROUP BY CUBE, and HAVING aren't allowed in queries that use TYPEOF.

Combining TYPEOF and Type

You can combine TYPEOF and Type in a SELECT statement. The following SELECT statement uses both TYPEOF and Type to filter the query and refine the set of returned fields based on the What field in Event.

```
SELECT Id,
TYPEOF What
WHEN Account THEN Phone
WHEN Opportunity THEN Amount
END
FROM Event
WHERE What.Type IN ('Account', 'Opportunity')
```

At run time, this SELECT statement always returns the ID for an Event, and then either Account. Phone, or Opportunity. Amount, depending on the object type referenced by the Event's What field. Note that no ELSE clause has been provided. Since this statement filters based on the What field in the WHERE clause, only Events that reference either an Account or Opportunity are returned, so the ELSE clause is not needed. If an ELSE clause was included in this case, it would be ignored at run time.

Object Types in WSDLs

In the Enterprise and Tooling API WSDLs, the object type of a polymorphic field depends on the version of the API.

1. In API version 46.0 and later (and in versions where the Developer Preview part of the SOQL Polymorphism feature is enabled), the object type is sObject. For example:

2. In other versions, the type is Name. For example:

This affects Java code generated from these WSDLs. For example, in Task.java, the Owner field is now defined as:

```
private com.sforce.soap.enterprise.sobject.SObject Owner;
```

For more information about generating Java code from WSDL files, see Setting Up Your Java Developer Environment.

Understanding Relationship Query Limitations

When you design SOQL relationship queries, there are several limitations to consider.

- Relationship queries are not the same as SQL joins. You must have a relationship between objects to create a join in SQQL.
- No more than 35 child-to-parent relationships can be specified in a query. A custom object allows up to 25 relationships, so you can reference all the child-to-parent relationships for a custom object in one query.
- No more than 20 parent-to-child relationships can be specified in a guery.
- In each specified relationship, no more than five levels can be specified in a child-to-parent relationship. For example, Contact.Account.Owner.FirstName (three levels).
- In each specified relationship, only one level of parent-to-child relationship can be specified in a query. For example, if the FROM clause specifies Account, the SELECT clause can specify only the Contact or other objects at that level. It could not specify a child object of Contact.
- You can query notes and attachments to get information about them, but you can't filter on the body of the note or attachment. You can't filter against the content of textarea fields, blobs, or Scontrol components in any object. For example, this query is valid, and it returns all account names and the owner ID for any notes associated with the account.

```
SELECT Account.Name, (SELECT Note.OwnerId FROM Account.Notes) FROM Account
```

However, this query isn't valid, because it attempts to evaluate information stored in the body of the note.

```
SELECT Account.Name, (SELECT Note.Body FROM Account.Notes WHERE Note.Body LIKE 'D%') FROM Account
```

If you remove the WHERE clause, the query is valid and returns the contents of the body of the note.

```
SELECT Account.Name, (SELECT Note.Body FROM Account.Notes) FROM Account
```

- Consider these limitations for external objects.
 - A subquery that involves external objects can fetch up to 1,000 rows of data.
 - Each SOQL query can have up to 4 joins across external objects and other types of objects.
 Each join requires a separate round trip to the external system when executing the query. Expect longer response times for each join in a query.
 - External objects don't support the ORDER BY clause in relationship queries. This limit applies only when the external data is
 accessed via the OData 2.0 adapter for Salesforce Connect.
 - If the primary or "driving" object for a SELECT statement is an external object, queryMore () supports only that primary object and doesn't support subqueries.

Using Relationship Queries with History Objects

Custom objects and some standard objects have an associated history object that tracks changes to an object record. You can use SOQL relationship queries to traverse a history object to its parent object.

For example, the following query returns every history row for Foo c and displays the name and custom fields of Foo:

```
SELECT OldValue, NewValue, Parent.Id, Parent.name, Parent.customfield_c FROM foo_history
```

This example query returns every Foo object row together with the corresponding history rows in nested subqueries:

```
SELECT Name, customfield_c, (SELECT OldValue, NewValue FROM Histories)
FROM foo_c
```

Using Relationship Queries with Data Category Selection Objects

Data categories are used to classify records. In SOQL, you can use the *Article*__DataCategorySelection or QuestionDataCategorySelection objects. You can also build a relationship guery with the DataCategorySelections relationship name in a FROM clause.

Imagine an Offer article type. The following query returns the ID of any categorization associated with an offer and the ID of the categorized article.

```
SELECT Id, ParentId
FROM Offer__DataCategorySelection
```

The following example uses the DataCategorySelections relationship name to build a relationship query that returns the ID of published offers and the ID of all the categorizations associated to these offers.

```
SELECT Id, Title
(
SELECT Id
FROM DataCategorySelections
```

```
)
FROM Offer__kav WHERE publishStatus='online';
```

Using Relationship Queries with the Partner WSDL

The partner WSDL doesn't contain the detailed type information that's available in the enterprise WSDL which you need for a relationship SOQL query. You must first execute a describeSObjects() call, and from the results, gather the information you need to create your relationship query:

- The relationshipName value for one-to-many relationships, for example, in an Account object, the relationship name for the
 asset child is Assets.
- The reference fields available for the relevant object, for example, whold, whatld, or ownerld on a Lead, Case, or custom object.

For an example of using the partner WSDL with relationship queries, see the examples on developer.salesforce.com (requires login).

Change the Batch Size in Queries

You can change the batch size (the number of rows that are returned in the query result object) that's returned in a query () or queryMore () call from the default 500 rows.

WSC clients can set the batch size by calling <code>setQueryOptions()</code> on the connection object. C# client applications can change this setting by specifying the batch size in the call QueryOptions portion of the SOAP header before invoking the <code>query()</code> call. The maximum batch size is 2,000 records. However this setting is only a suggestion. There is no guarantee that the requested batch size will be the actual batch size. This is done to maximize performance.



Note: The batch size will be no more than 200 if the SOQL statement selects two or more custom fields of type long text. This is to prevent large SOAP messages from being returned.

The following sample Java (WSC) code demonstrates setting the batch size to two hundred and fifty (250) records.

```
public void queryOptionsSample() {
  connection.setQueryOptions(250);
}
```

The following sample C# (.NET) code demonstrates setting the batch size to two hundred and fifty (250) records.

```
private void queryOptionsSample()
{
   binding.QueryOptionsValue = new QueryOptions();

   binding.QueryOptionsValue.batchSize = 250;
   binding.QueryOptionsValue.batchSizeSpecified = true;
}
```

SOQL Limits on Objects

SOQL applies specific limits to objects and situations in search results. SOQL limits are defined for the ContentDocumentLink object, ContentHubItem object, big objects, external objects, NewsFeed, KnowledgeArticleVersion, RecentlyViewed, TopicAssignment, UserRecordAccess, UserProfileFeed, and Vote.

Some objects or situations have specific limits on SOQL.

Object	Description	
ContentDocumentLink	A SOQL query must filter on one of Id, ContentDocumentId, or LinkedEntityId.	
ContentHubItem	A SOQL query must filter on one of Id, ExternalId, or ContentHubRepositoryId.	
Custom metadata types	Custom metadata types support the following SOQL query syntax.	
	SELECT fieldList [] FROM objectType [USING SCOPE filterScope] [WHERE conditionExpression] [ORDER BY field {ASC DESC} [NULLS {FIRST LAST}]]	
	 You can use metadata relationship fields in the fieldList and conditionExpression. FROM can include only 1 object. You can use the following operators. IN and NOT IN =, >, >=, <, <=, and != LIKE, including wild cards AND 	
	 You can use ORDER BY only with non-relationship fields. You can use ORDER BY, ASC, and DESC with multiple (non-relationship) fields. You can only use ORDER BY when the ordered field is a selected field. Metadata relationship fields support all standard relationship queries. 	
Big objects	 A SOQL query can only filter on the fields defined in the big object's index, in the order that they are defined, without gaps. The last field in a query can use only the following operators. =, <. >, <=, >=, and IN Any prior fields in a query can use only the = operator. Big objects don't support the following operators. !=, LIKE, NOT IN, EXCLUDES, and INCLUDES 	
External objects	 A subquery that involves external objects can fetch up to 1,000 rows of data. Each SOQL query can have up to 4 joins across external objects and other types of objects. Each join requires a separate round trip to the external system when executing the query. Expect longer response times for each join in a query. External objects don't support the following aggregate functions and clauses. AVG() function COUNT (fieldName) function (however, COUNT() is supported) HAVING clause 	

Object Description

- GROUP BY clause
- MAX () function
- MIN() function
- SUM() function
- External objects also don't support the following.
 - EXCLUDES operator
 - FOR VIEW clause
 - FOR REFERENCE clause
 - INCLUDES operator
 - LIKE operator
 - toLabel() function
 - TYPEOF clause
 - with clause

The following limits apply only to the OData 2.0 and 4.0 adapters for Salesforce Connect.

- External objects have the following limitations for the ORDER BY clause.
 - NULLS FIRST and NULLS LAST are ignored.
 - External objects don't support the ORDER BY clause in relationship queries.
- The COUNT () aggregate function is supported only on external objects whose external data sources have Request Row Counts enabled. Specifically, the response from the external system must include the total row count of the result set.

The following limits apply only to custom adapters for Salesforce Connect.

- Location-based SOQL gueries of external objects aren't supported.
- If a SOQL guery of an external object includes the following, the guery fails.
 - convertCurrency() function
 - UPDATE TRACKING clause
 - UPDATE VIEWSTAT clause
 - USING SCOPE clause
- In an ORDER BY clause, the following are ignored.
 - NULLS FIRST syntax
 - NULLS LAST syntax
- In Apex tests, use dynamic SOQL to query external objects. Tests that perform static SOQL queries of external objects fail.

The following limits apply only to external objects associated with a SharePoint 2010/2013 external data source using SecureAgent.

• In SOQLueries of external objects, In clauses with more than approximately 15 IDs return the error "This operation is too complicated for Secure Agent." The exact In clause limit varies based on SharePoint ID length.

Object	Description	
KnowledgeArticleVersion	• Always filter on a single value of PublishStatus unless the query filters on one or more primary key IDs. To support security, only users with the "Manage Articles" permission see articles whose PublishStatus value is Draft.	
	 Archived article versions are stored in the articletype_kav object. To query archived article versions, specify the article Id and set IsLatestVersion='0'. 	
	 Always filter on a single value of Language. However, in SOQL, you can filter on more than one Language if there is a filter on Id or KnowledgeArticleId. 	
	 You cannot use binding variables in Apex SOQL statements with KnowledgeArticleVersion objects. For example, the following SOQL statement causes a compilation error. 	
	<pre>final String PUBLISH_STATUS_ONLINE = 'Online'; List<knowledgekav> articles = [SELECT Id FROM Knowledgekav WHERE PublishStatus = :PUBLISH_STATUS_ONLINE];</knowledgekav></pre>	
	Instead, use dynamic SOQL as follows. See Dynamic SOQL in Apex Developer Guide.	
	<pre>final String PUBLISH_STATUS_ONLINE = 'Online'; final String q = 'SELECT Id, PublishStatus FROM Knowledgekav WHERE PublishStatus = :PUBLISH_STATUS_ONLINE'; List<knowledgekav> articles = Database.query(q);</knowledgekav></pre>	
NewsFeed	 No SOQL limit if logged-in user has "View All Data" permission. If not, specify a LIMIT clause of 1,000 records or fewer. 	
	• SOQL ORDER BY on fields using relationships is not available. Use ORDER BY on fields on the root object in the SOQL query.	
RecentlyViewed	The RecentlyViewed object is updated every time the logged-in user views or references a record. It is also updated when records are retrieved using the FOR VIEW or FOR REFERENCE clause in a SOQL query. To ensure that the most recent data is available, RecentlyViewed data is periodically truncated down to 200 records per object. RecentlyViewed data is retained for 90 days, after which it is removed on a periodic basis.	
TopicAssignment	No SOQL limit if logged-in user has "View All Data" permission. If not, do one of the following:	
	Specify a LIMIT clause of 1,100 records or fewer.	
	 Filter on Id or Entity when using a WHERE clause with "=". 	
UserRecordAccess	Always use the query formats specified in the SOAP API Developer Guide.	
	 May include an ORDER BY clause. You must ORDER BY HasAccess if you SELECT HasAccess, and ORDER BY MaxAccessLevel if you SELECT MaxAccessLevel. 	
	 Maximum number of records that can be queried is 200. 	

Object	Description
UserProfileFeed	 No SOQL limit if logged-in user has "View All Data" permission. If not, specify a LIMIT clause of 1,000 records or fewer.
	 SOQL ORDER BY on fields using relationships is not available. Use ORDER BY on fields on the root object in the SOQL query.
	Also, a SOQL query must include WITH UserId = {userId}.
Vote	• ParentId = [single ID]
	• Parent.Type = [single type]
	• Id = [single ID]
	• Id IN = [list of IDs]

SOQL with Big Objects

You can guery the fields in a big object's index using a subset of standard SOQL commands.

Build an index query starting from the first field defined in the index, without gaps between the first and last field in the query. You can use =, <, >, <=, or >=, or IN on the last field in your query. Any prior fields in your query can only use the = operator. The !=, LIKE, NOT IN, EXCLUDES, and INCLUDES operators are not valid in any query.

You can include the system fields CreatedById, CreatedDate, and SystemModstamp in queries. To retrieve a list of results, do not use the Id field in a query. Including Id in a query returns only results that have an empty ID (0000000000000000 or 0000000000000AAA).



Note: When you use Developer Console to generate a query from a resource, the Id field is included automatically. To query big objects in Developer Console, remove Id from the generated query.

The following queries assume that you have a table in which the index is defined by LastName__c, FirstName__c, and PhoneNumber__c.

This query specifies all three fields in the index. In this case, the filter on PhoneNumber c can be a range.

```
SELECT LastName__c, FirstName__c, PhoneNumber__c
FROM Phone_Book__b
WHERE LastName__c='Kelly' AND FirstName__c='Charlie' AND PhoneNumber__c='2155555555'
```

This query specifies only the first two fields in the index. In this case, the filter on FirstName c can be a range.

```
SELECT LastName__c, FirstName__c, PhoneNumber__c
FROM Phone_Book_ b
WHERE LastName__c='Kelly' AND FirstName__c='Charlie'
```

This query specifies only the first field in the index. The filter on LastName c can be a range.

```
SELECT LastName__c, FirstName__c, PhoneNumber__c
FROM Phone_Book__b
WHERE LastName__c='Kelly'
```

This query doesn't work because of a gap in the query where FirstName c is required.

```
SELECT LastName__c, FirstName__c, PhoneNumber__c
FROM Phone_Book__b
WHERE LastName__c='Kelly' AND PhoneNumber__c='2155555555'
```



Note: These restrictions do not apply to Async SOQL queries against big objects.

Syndication Feed SOQL and Mapping Syntax

Syndication feed services use a SOQL query and mapping specification that allows applications to point to sets of objects and individual objects and to traverse relationships between objects. Several options can be added as query string parameters to filter and control how the data is presented. Syndication feeds can be defined for public sites.

For full information about the limitations on SOQL in query feed definitions, see the Salesforce online help for syndication feeds.

Location-Based SOQL Queries

Location-based SOQL queries let you compare and query location values stored in Salesforce. You can calculate the distance between two location values, such as between a warehouse and a store. Or you can calculate the distance between a location value and fixed latitude-longitude coordinates, such as between a warehouse and 37.775°, -122.418°—also known as San Francisco.

The geolocation custom field type allows you to create a field to store location values. A geolocation field identifies a location by its latitude and longitude. Standard addresses on Salesforce objects also include a geolocation field that, when populated, can be used in similar ways—with a few restrictions. You can compare and query the locations of both types, for example, to find the 10 closest accounts.

For more information and considerations to keep in mind, see "Compound Fields" in the SOAP API Developer's Guide.

Field Types That Support Location-Based SOQL Queries

SOQL supports using simple numeric values for latitude and longitude using the GEOLOCATION function. These values can come from standard numeric fields, user input, calculations, and so on. They can also come from the individual components of a geolocation field, which stores both latitude and longitude in a single logical field. If a geocoding service has populated the geolocation field of a standard address, you can also use latitude and longitude values directly from an address.

SOQL queries made using the SOAP and REST APIs also support using geolocation fields, including address fields that have a geolocation component, directly in SOQL statements. This often results in simpler SOQL statements. Compound fields can *only* be used in SOQL queries made through the SOAP and REST APIs.

SELECT Clause

Retrieve records with locations saved in geolocation or address fields as individual latitude and longitude values by appending "__latitude__s" or "__longitude__s" to the field name, instead of the usual "__c". For example:

```
SELECT Name, Location__latitude__s, Location__longitude__s
FROM Warehouse__c
```

This query finds all the warehouses that are stored in the custom object Warehouse. The results include each warehouse's latitude and longitude values individually. To select the latitude and longitude components individually, use separate field components for Location__c.

SOQL executed using the SOAP or REST APIs can SELECT the compound field, instead of the individual elements. Compound fields are returned as structured data rather than primitive values. For example:

```
SELECT Name, Location__c
FROM Warehouse__c
```

This query retrieves the same data as the previous query, but the Location_c field is a compound geolocation field, and the results combine the two primitive values. Here are sample results from a REST API request.

```
"totalSize" : 10,
"done" : true,
"records" : [ {
 "attributes" : {
   "type" : "Warehouse c",
   "url" : "/services/data/v30.0/sobjects/Warehouse c/a06D0000001704nIAE"
 "Name" : "Ferry Building Depot",
 "Location c" : {
   "latitude" : 37.79302,
   "longitude" : -122.394507
 }
}, {
  "attributes" : {
   "type" : "Warehouse c",
    "url" : "/services/data/v30.0/sobjects/Warehouse c/a06D0000001704oIAE"
 },
 "Name" : "Aloha Warehouse",
 "Location c" : {
   "latitude" : 37.786108,
   "longitude" : -122.430152
 }
},
. . .
```

WHERE Clause

Retrieve records with locations within or outside of a certain radius with distance conditions in the WHERE clause of the query. To construct an appropriate distance condition, use the following functions.

DISTANCE

Calculates the distance between two locations in miles or kilometers.

Usage: DISTANCE (*mylocation1*, *mylocation2*, 'unit') and replace *mylocation1* and *mylocation2* with two location fields, or a location field and a value returned by the GEOLOCATION function. Replace *unit* with mi (miles) or km (kilometers).

GEOLOCATION

Returns a geolocation based on the provided latitude and longitude. Must be used with the DISTANCE function.

Usage: GEOLOCATION (*latitude*, *longitude*) and replace *latitude* and *longitude* with the corresponding geolocation, numerical code values.

Compare two field values, or a field value with a fixed location. For example:

```
SELECT Name, Location__c
FROM Warehouse__c
WHERE DISTANCE(Location__c, GEOLOCATION(37.775,-122.418), 'mi') < 20</pre>
```

ORDER BY Clause

Sort records by distance using a distance condition in the ORDER BY clause. For example:

```
SELECT Name, StreetAddress_c
FROM Warehouse_c
WHERE DISTANCE(Location_c, GEOLOCATION(37.775,-122.418), 'mi') < 20
ORDER BY DISTANCE(Location_c, GEOLOCATION(37.775,-122.418), 'mi')
LIMIT 10
```

This query finds up to 10 of the warehouses in the custom object Warehouse that are within 20 miles of the geolocation 37.775°, -122.418°, which is San Francisco. The results include the name and address of each warehouse, but not its geocoordinates. The nearest warehouse is first in the list. The farthest location is last.

How SOQL Treats Null Location Values

Geolocation fields are compound fields that combine latitude and longitude values to describe a specific point on Earth. Null values are valid only if *both* latitude and longitude are null.

A record that has an invalid geolocation field value is treated as though both values are null when used in SOQL WHERE DISTANCE () and ORDER BY clauses. A record that has a geolocation field in which either the latitude or longitude is null is treated as though the field has not been set.

When a compound geolocation field is used in a SELECT clause, invalid geolocation values return null. For example:

```
SELECT Name, Location__c
FROM Warehouse__c
LIMIT 5
```

Values such as the ones in this table are returned from API calls.

Name	Locationc
Ferry Building Depot	null
Aloha Warehouse	(37.786108,-122.430152)
Big Tech Warehouse	null
S H Frank & Company	null
San Francisco Tech Mart	(37.77587,-122.399902)

These results include three null geolocation values. It's not possible to tell which values are genuinely null, and which are invalid data.

When the individual field components of that same geolocation field are used in a SELECT clause, the saved values are returned as before. Non-null values are returned as that value, and null values return as null. For example:

```
SELECT Name, Location__latitude__s, Location__longitude__s
FROM Warehouse__c
LIMIT 5
```

These values are sample query results.

Name	Locationlatitudes	Locationlongitudes
Ferry Building Depot	null	-122.394507
Aloha Warehouse	37.786108	-122.430152
Big Tech Warehouse	null	null
S H Frank & Company	37.763662	null
San Francisco Tech Mart	37.77587	-122.399902

In these results, only one geolocation field is genuinely null. The other two, with partial nulls, are invalid.

When you create formula fields that you plan to use for DISTANCE calculations, select **Treat blank fields as blanks** in the Blank Field Handling section. If you select **Treat blank fields as zeros**, distances are calculated from 0°, 0°—the point where the equator intersects the prime meridian—when your geolocation fields have null values. On record detail pages, null geolocation values in DISTANCE formula fields that are set to **Treat blank fields as zeros** cause the formula fields to display as empty.

How SOQL Calculates and Compares Distances

The DISTANCE function approximates the haversine, or "great circle," distance calculation within 0.0002%. This formula assumes that the Earth is a perfect sphere, when in fact it's an ellipsoid: an irregular one. Errors from this assumption can be up to 0.55% crossing the equator, but are usually below 0.3%, depending on latitude and direction of travel.

The DISTANCE function is fine for calculating the 10 stores closest to a customer's current location. But don't fuel your plane for a flight from San Francisco to Sydney based on it.

Another implication of this approximation is that geolocations and distances have no notion of "equal." You can't check locations or distances for equality. You can only determine whether one location is farther away or closer than another location, or one distance is greater or smaller than another. To verify that two locations are "the same," treat their distance as a floating point number and compare the difference to a tolerance value. For example, this WHERE clause finds other records within 25 feet of testLocation.

```
WHERE ( DISTANCE(Location__c, testLocation) < 0.05 )
```

Although the errors are small for nearly identical distances, the errors can cause a location query to include or exclude expected locations. If your application requires precise distance calculations and comparisons, we recommend that you do your own math.

Location-Based SOQL Query Considerations

Location-based queries are supported in SOQL in Apex and in the SOAP and REST APIs. Keep in mind these considerations.

- DISTANCE and GEOLOCATION are supported in WHERE and ORDER BY clauses in SOQL, but not in GROUP BY. DISTANCE is supported in SELECT clauses.
- DISTANCE supports only the logical operators > and <, returning values within (<) or beyond (>) a specified radius.

- When using the GEOLOCATION function in SOQL queries, the geolocation field must precede the latitude and longitude coordinates. For example, DISTANCE (warehouse_location__c, GEOLOCATION(37.775,-122.418), 'km') works but DISTANCE (GEOLOCATION(37.775,-122.418), warehouse location c, 'km') doesn't work.
- Apex bind variables aren't supported for the units parameter in DISTANCE or GEOLOCATION functions. This query doesn't
 work.

```
String units = 'mi';
List<Account> accountList =
   [SELECT ID, Name, BillingLatitude, BillingLongitude
   FROM Account
   WHERE DISTANCE(My_Location_Field_c, GEOLOCATION(10,10), :units) < 10];</pre>
```

For more information, see "Compound Field Considerations and Limitations" in the SOAP API Developer's Guide.

Async SOQL

Async SOQL is a method for running SOQL queries when you can't wait for immediate results. These queries are run in the background over Salesforce big object data. Async SOQL provides a convenient way to query large amounts of data stored in Salesforce.

Async SOQL is implemented as a RESTful API that enables you to run queries in the familiar syntax of SOQL. Because of its asynchronous operation, you can subset, join, and create more complex queries and not be subject to timeout limits. This situation is ideal when you have millions or billions of records and need more performant processing than is possible using synchronous SOQL. The results of each query are deposited into an object you specify, which can be a standard object, custom object, or big object.

The limit for Async SOQL queries is one concurrent query at a time.

Async SOQL Versus SOQL

SOQL and Async SOQL provide many of the same capabilities. So when would you use an Async SOQL query instead of standard SOQL?

Use standard SOOL when:

- You want to display the results in the UI without having the user wait for results.
- You want results returned immediately for manipulation within a block of Apex code.
- You know that the guery returns a small amount of data.

Use Async SOQL when:

- You are querying against millions of records.
- You want to ensure that your query completes.
- You do not need to do aggregate queries or filtering outside of the index.

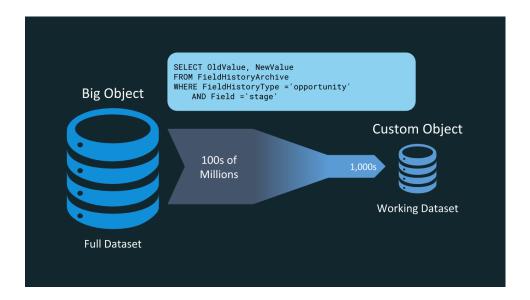
Use Case: Create a Working Dataset with Filtering

EDITIONS

Available in: both Salesforce Classic and Lightning Experience

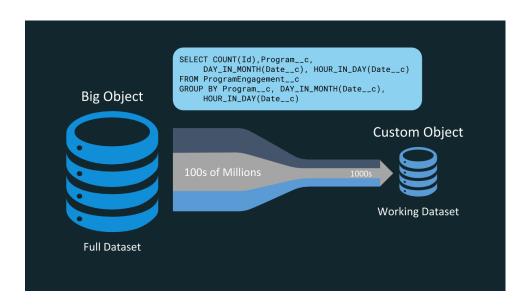
Available in: **Enterprise**, **Performance**, **Unlimited**, and **Developer** Editions for up to 1 million records

Extra record capacity and Async SOQL query available as an add-on license.



For example, let's say that you want to analyze the years and years of opportunity history collected by Salesforce. The results could help you identify which current and future opportunities are more likely to close and give you a better picture of your forecast. But because the opportunity history data is stored with all the field history data across the application, the volume of data is too large to query directly. That's where Async SOQL comes in! You can use it to write a query that extracts a smaller, representative subset of the data that you're interested. You can store this working dataset in a custom object and use it in reports, dashboards, or any other Lightning Platform feature.

Use Case: Create a Working Dataset with Coarse Aggregations



With big objects, you can now bring a much finer level of detail into your applications using data that you already have. For example, every interaction an individual has with your marketing campaign is stored as data that you can use, but it's unwieldy in its raw form. Async SOQL allows you to aggregate that data by campaign and day and to extract the relevant details of the full dataset into a smaller, usable dataset. As in the previous example, the smaller working set can live in a custom object and be used in your reports and dashboards.

CHAPTER 3 Salesforce Object Search Language (SOSL)

In this chapter ...

- Typographical Conventions in This Document
- SOSL Limits
- SOSL Limits on External Objects
- SOSL Syntax
- Example Text Searches
- convertCurrency()
- FIND {SearchQuery}
- FORMAT()
- IN SearchGroup
- LIMIT n
- OFFSET n
- ORDER BY Clause
- RETURNING FieldSpec
- toLabel(fields)
- Update an Article's Keyword Tracking with SOSL
- Update an Article's Viewstat with SOSL
- USING Listview=
- WHERE conditionExpression
- WITH DATA CATEGORY DataCategorySpec
- WITH DivisionFilter
- WITH HIGHLIGHT
- WITH METADATA
- WITH NETWORK NetworkldSpec
- WITH PricebookId
- WITH SNIPPET

Use the Salesforce Object Search Language (SOSL) to construct text-based search queries against the search index.

You can search text, email, and phone fields for multiple objects, including custom objects, that you have access to in a single guery in the following environments.

- SOAP or REST calls
- Apex statements
- Visualforce controllers and getter methods
- Schema Explorer of the Eclipse Toolkit



Note: If your org has relationship queries enabled, SOSL supports SOQL relationship queries.

When to Use SOSL

Use SOSL when you don't know which object or field the data resides in, and you want to:

- Retrieve data for a specific term that you know exists within a field. Because SOSL can tokenize
 multiple terms within a field and build a search index from this, SOSL searches are faster and can
 return more relevant results.
- Retrieve multiple objects and fields efficiently where the objects might or might not be related to one another.
- Retrieve data for a particular division in an organization using the divisions feature.
- Retrieve data that's in Chinese, Japanese, Korean, or Thai. Morphological tokenization for CJKT terms helps ensure accurate results.

Performance Considerations

If your searches are too general, they are slow and return too many results. Use the following clauses to define efficient text searches.

- IN: Limits the types of fields to search, including email, name, or phone.
- LIMIT: Specifies the maximum number of rows to return.
- OFFSET: Displays the search results on multiple pages.
- RETURNING: Limits the objects and fields to return.
- WITH DATA CATEGORY: Specifies the data categories to return.
- WITH DivisionFilter: Specifies the division field to return.
- WITH NETWORK: Specifies the community ID to return.
- WITH PricebookId: Specifies the price book ID to return.

• WITH SPELL_CORRECTION

Navigating This Document

- To see a list of available resources, see SOSL Syntax.
- To get started working with SOSL, see Example Text Searches.

Typographical Conventions in This Document

This SOSL reference uses specific typographical conventions.

Use the following typographical conventions:

Convention	Description
FIND Name IN Account	Courier font indicates items that you should type as shown. In a syntax statement, Courier font also indicates items that you should type as shown, except for curly braces, square brackets, ellipsis, and other typographical markers explained in this table.
FIND fieldname IN objectname	Italics represent a variable or placeholder. You supply the actual value.
1	The pipe character separates alternate elements. For example, in the clause UPDATE TRACKING VIEWSTAT [,], the character indicates that you can use either TRACKING or VIEWSTAT after UPDATE.
[LIMIT n]	Square brackets indicate an optional element. For example, <code>[LIMIT n]</code> means that you can specify a <code>LIMIT</code> clause. Don't type square brackets as part of a SOSL command. Nested square brackets indicate elements that are optional and can only be used if the parent optional element is present. For example, in the clause <code>[ORDER BY fieldname [ASC DESC] [NULLS {FIRST LAST}]]</code> , ASC, <code>DESC</code> , or the <code>NULLS</code> clause cannot be used without the <code>ORDER BY</code> clause.
[] and [,]	Square brackets containing an ellipsis indicate that the preceding element can be repeated up to the limit for the element. If a comma is also present, the repeated elements must be separated by commas. If the element is a list of choices grouped with curly braces, you can use items from the list in any order. For example, in the clause [[UPDATE [TRACKING VIEWSTAT] [,]], the [,] indicates that you can use TRACKING, VIEWSTAT, or both:
	UPDATE TRACKING
	UPDATE VIEWSTAT
	UPDATE TRACKING, VIEWSTAT

SOSL Limits

The search engine limits the number of records analyzed at each stage of the search process. Sometimes, these limits cause a matching record to be excluded from a user's results.

This image illustrates how the search engine processes SOSL searches and limits results. Each color represents an object, and each raindrop represents some records. The numbers correspond to this flow:

- 1. The search engine looks for matches to the search term across a maximum of 2,000 records (this limit starts with API version 28.0)
- 2. SOSL applies different limits for a given object or situation. If the search is for a single object, the full record limit is applied. If the search is global across multiple objects, each object has individual limits that total 2,000 records.
- **3.** Admins (users with the View All Data permission) see the full set of results returned.

4. For all other users, SOSL applies user permission filters. Individual users see only those records that they have access to. Results sets and order vary by the user issuing the search and can change throughout the day as records are added or removed from the index.



Example: Joe Smith, a sales executive at Acme, Inc., wants to find the account record for Industrial Computing. He types Industrial into the search bar. Because so many records match the search term Industrial, a limit is imposed on the results. Unfortunately for Joe, the record he wanted is outside the limit. This concept is illustrated in the image as the single raindrop outside of the filter. Because Joe used a global search, limits are applied to each object type to make up the 2,000 record limit. The illustration shows five blue raindrops going into the filter, but only three make it to the next stage. If Joe limited his search to just one object, the limit would apply to only that object, increasing the chance that the record he wanted would be returned. Joe retries his search by typing Industrial Computing San Francisco. With a more specific search term, the search engine is able to return better matches, even with the same limits applied. In this scenario, the record Joe's looking for is one of the blue raindrops that makes it from the top of the filter all the way through to Joe's search results page.

SOSL Limits on External Objects

SOSL applies specific limits to external objects in search results.

- To include an external object in SOSL and Salesforce searches, enable search on both the external object and the external data source. However, syncing always overwrites the external object's search status to match the search status of the external data source.
- Only text, text area, and long text area fields on external objects can be searched. If an external object has no searchable fields, searches on that object return no records.
- External objects don't support the following.
 - INCLUDES operator
 - LIKE operator
 - EXCLUDES operator
 - toLabel() function
- External objects also don't support Salesforce Knowledge-specific clauses, including the following.
 - UPDATE TRACKING clause
 - UPDATE VIEWSTAT clause
 - WITH DATA CATEGORY clause
- External objects must be specified explicitly in a RETURNING clause to be returned in search results. For example:

```
FIND {MyProspect} RETURNING MyExternalObject, MyOtherExternalObject
```

• Text strings must be 100 or fewer characters.

The following limits apply only to the OData 2.0 and 4.0 adapters for Salesforce Connect.

• The OData adapters for Salesforce Connect don't support logical operators in a FIND clause. We send the entire search query string to the external system as a case-sensitive single phrase after removing all ASCII punctuation characters except hyphens (-). For example, FIND {MyProspect OR "John Smith"} searches for the exact phrase "MyProspect OR John Smith".

The following limits apply only to custom adapters for Salesforce Connect.

- The convertCurrency() function isn't supported in SOSL queries of external objects.
- WITH clauses aren't supported in SOSL queries of external objects.

SOSL Syntax

A SOSL query begins with the required FIND clause. You can then add optional clauses to filter the query by object type, fields, data categories, and more. You can also determine what is returned. For example, you can specify the order of the results and how many rows to return.

After the required FIND clause, you can add one or more optional clauses in the following order.

```
FIND {SearchQuery}
[ IN SearchGroup ]
[ RETURNING FieldSpec [[ toLabel(fields)] [convertCurrency(Amount)] [FORMAT()]] ]
[ WITH DivisionFilter ]
[ WITH DATA CATEGORY DataCategorySpec ]
[ WITH SNIPPET[(target_length=n)] ]
[ WITH NETWORK NetworkIdSpec ]
```

```
[ WITH PricebookId ]
[ WITH METADATA ]
[ LIMIT n ]
[ UPDATE [TRACKING], [VIEWSTAT] ]
```



Mote: Offset in and where conditionExpression are included within Returning FieldSpec.

where:

Syntax	Description
convertCurrency()	Optional. If an org has multicurrency enabled, converts currency fields to the user's currency.
FIND {SearchQuery}	Required. Specifies the text (words or phrases) to search for. Enclose the search query with curly braces.
	If the SearchQuery string is longer than 10,000 characters, no result rows are returned. If SearchQuery is longer than 4,000 characters, any logical operators are removed. For example, the AND operator in a statement with a SearchQuery that's 4,001 characters will default to the OR operator, which could return more results than expected.
FORMAT()	Optional. Use FORMAT with the FIND clause to apply localized formatting to standard and custom number, date, time, and currency fields. The FORMAT function supports aliasing. In addition, aliasing is required when the query includes the same field multiple times.
IN SearchGroup	Optional. Scope of fields to search. One of the following values:
	• ALL FIELDS
	• NAME FIELDS
	• EMAIL FIELDS
	• PHONE FIELDS
	• SIDEBAR FIELDS
	If unspecified, the default is ALL FIELDS. You can specify the list of objects to search in the RETURNING <i>FieldSpec</i> clause.
	Note: This clause doesn't apply to articles, documents, feed comments, feed items, files, products, and solutions. If these objects are specified in the RETURNING clause, the search is not limited to specific fields, and all fields are searched.
LIMIT n	Optional. Specifies the maximum number of rows to return in the text query, up to 2,000. If unspecified, the default is 2,000, the largest number of rows that can be returned. These limits apply to API version 28.0 and later. Previous versions support a maximum of 200 rows.

Syntax	Description
OFFSET n	Optional. When expecting many records in a query's results, you can display the results in multiple pages by using the OFFSET clause in a SOSL query. For example, you can use OFFSET to display records 51–75 and then jump to displaying records 301–350. Using OFFSET is an efficient way to handle large results sets.
ORDER BY	Optional. Specifies the order in which search results are returned using the ORDER BY clause. You can also use this clause to display empty records at the beginning or end of the results.
RETURNING FieldSpec	Optional. Information to return in the search result. List of one or more objects and, within each object, list of one or more fields, with optional values to filter against. If unspecified, the search results contain the IDs of all objects found.
toLabel(fields)	Optional. Results from a query are returned translated into the user's language.
USING ListView=	Optional clause used to search within a single given object's list view. Only one list view can be specified. Only the first 2,000 records of the list view are searched, according to the sort order the user has set for the list view.
[UPDATE [TRACKING VIEWSTAT][,]]]	 Optional. If an org uses Salesforce Knowledge: UPDATE TRACKING tracks keywords used in Salesforce Knowledge article search. Update an Article's Viewstat with SOSL updates an article's view statistics. UPDATE TRACKING, VIEWSTAT does both.
WHERE conditionExpression	Optional. By default, a SOSL query on an object retrieves all rows that are visible to the user. To limit the search, filter the search result by specific field values.
WITH DATA CATEGORY DataCategorySpec	Optional. If an org uses Salesforce Knowledge articles or answers, filters all search results based on one or more data categories.
WITH DivisionFilter	Optional. If an org uses divisions, filters all search results based on values for the Division field.
WITH HIGHLIGHT	Optional. In search results for certain objects, highlights the terms matching the search query.
WITH METADATA = MetadataSpec	Optional. Specifies if metadata is returned in the response. The default setting is no, meaning no metadata is returned.
WITH NETWORK NetworkIdSpec	Optional. Filters search results by community ID.
WITH PricebookId	Optional. Filters product search results by a single price book ID.

Syntax	Description
<pre>WITH SNIPPET [(target_length=n)]</pre>	Optional. If an org uses Salesforce Knowledge articles, displays contextual snippets and highlights the search term for each article in the search results. By default, each snippet displays up to about 300 characters, which is usually about three lines of text in a standard browser window. Add the optional target_length parameter to specify an alternate target length, which can be from 100 and 500 characters.
WITH SPELL_CORRECTION	Optional. When set to true, spell correction is enabled for searches that support spell correction. When set to false, spell correction is not enabled. The default value is true.



Note: The SOSL statement character limit is tied to the SOQL statement character limit defined for your org. By default, SOQL and SOSL queries cannot exceed 20,000 characters. For SOSL statements that exceed this maximum length, the API returns a MALFORMED_SEARCH exception code, and no result rows are returned.

Example Text Searches

The following are examples of text searches that use SOSL.

Look for joe anywhere in the system. Return the IDs of the records where joe is found.

```
FIND {joe}
```

Look for the name Joe Smith anywhere in the system, in a case-insensitive search. Return the IDs of the records where Joe Smith is found.

```
FIND {Joe Smith}
```

Look for the name Joe Smith in the name field of a lead, return the ID field of the records.

```
FIND {Joe Smith}
IN Name Fields
RETURNING lead
```

Look for the name Joe Smith in the name field of a lead and return the name and phone number.

```
FIND {Joe Smith}
IN Name Fields
RETURNING lead(name, phone)
```

Look for the name Joe Smith in the name field of a lead and return the name and phone number of any matching record that was also created in the current fiscal quarter.

```
FIND {Joe Smith}
IN Name Fields
RETURNING lead (name, phone Where createddate = THIS_FISCAL_QUARTER)
```

Look for the name Joe Smith or Joe Smythe in the name field of a lead or contact and return the name and phone number. If an opportunity is called Joe Smith or Joe Smythe, the opportunity should not be returned.

```
FIND {"Joe Smith" OR "Joe Smythe"}
IN Name Fields
RETURNING lead(name, phone), contact(name, phone)
```

Wildcards:

```
FIND {Joe Sm*}
FIND {Joe Sm?th*}
```

Delimiting "and" and "or" as literals when used alone:

```
FIND {"and" or "or"}
FIND {"joe and mary"}
FIND {in}
FIND {returning}
FIND {find}
```

Escaping special characters & $|!()\{\}[] \land " \sim *?: \ + -$

```
FIND {right brace \}}
FIND {asterisk \*}
FIND {question \?}
FIND {single quote \'}
FIND {double quote \"}
```



Note: Apex requires that you surround SOQL and SOSL statements with square brackets to use them on the fly. You can use Apex script variables and expressions when preceded by a colon (:).

convertCurrency()

Use convertCurrency () in a SOSL query to convert currency fields to the user's currency. This action requires that the organization is multi-currency enabled.

Use this syntax for the RETURNING clause:

convertCurrency (Amount)

For example,

```
FIND {test} RETURNING Opportunity(Name, convertCurrency(Amount))
```

If an org has enabled advanced currency management, dated exchange rates are used when converting currency fields on opportunities, opportunity line items, and opportunity history. With advanced currency management, convertCurrency uses the conversion rate that corresponds to a given field (for example, CloseDate on opportunities). When advanced currency management isn't enabled, the most recent conversion date entered is used.

You can't use the convertCurrency () function in a WHERE clause. If you do, an error is returned. Use the following syntax to convert a numeric value to the user's currency from any active currency in your org.

```
WHERE Object_name Operator ISO_CODEvalue
```

For example:

```
FIND {test} IN ALL FIELDS RETURNING Opportunity(Name WHERE Amount>USD5000)
```

In this example, opportunity records are returned if the record's currency Amount value is greater than the equivalent of USD5000. For example, an opportunity with an amount of USD5001 is returned, but not JPY7000.

Use an ISO code that your org has enabled and made active. If you don't put in an ISO code, the numeric value is used instead of comparative amounts. Using the previous example, opportunity records with JPY5001, EUR5001, and USD5001 would be returned. If you use IN in a WHERE clause, you can't mix ISO code and non-ISO code values.



Note: Ordering is always based on the converted currency value, just like in reports. Thus, convertCurrency() cannot be used with the ORDER BY *Clause*.

The currentCurrency () function supports aliasing. In addition, aliasing is required when the query includes the same field multiple times. For example:

FIND {Acme} RETURNING Account (Annual Revenue, convert Currency (Annual Revenue) Alias Currency)

FIND {SearchQuery}

Use the required FIND clause of a SOSL query to specify the word or phrase to search for. A search query includes the literal word or phrase and can also include wildcards and logical operators (AND, OR, and AND NOT).

A search query includes:

- The literal text (single word or a phrase) to search for
- Optionally, Wildcards
- Optionally, logical Operators, including grouping parentheses

Searches are evaluated from left to right and use Unicode (UTF-8) encoding. Text searches are case-insensitive. For example, searches for Customer, customer, and CUSTOMER return the same results.

You can't enter special types of text expressions (such as macros, functions, or regular expressions) that are evaluated at run time in the FIND clause.



Note: Surround the SearchQuery with curly brackets to distinguish the search expression from other clauses in the text search.

Search Terms

A SearchQuery can contain:

- Single words, such as test or hello
- Phrases that include words and spaces surrounded by double quotes, such as "john smith"

The search engine splits record information separated by spaces or punctuation into separate tokens. The search engine returns accurate search results from searches in East Asian languages that don't include spaces between words using morphological tokenization.



Example: Consider the problem of indexing the term "Tokyo Prefecture" and a subsequent search for Kyoto in Japanese.

Index	Search
東京都	京都
Tokyo Prefecture	Kyoto

Morphological tokenization segments the term 東京都 (Tokyo Prefecture) into two tokens.

東京	都
Tokyo	Prefecture

This form of tokenization ensures that a search for 京都 (Kyoto) returns only results that include 京都 (Kyoto) and not 東京都 (Tokyo Prefecture).

Wildcards

You can specify the following wildcard characters to match text patterns in your search:

Wildcard	Description	
*	Asterisks match zero or more characters at the middle or end of your search term. For example, a search for <i>john</i> finds items that start with <i>john</i> , such as, <i>john</i> , <i>johnson</i> , or <i>johnny</i> . A search for <i>mi* meyers</i> finds items with <i>mike</i> meyers or michael meyers.	
	If you are searching for a literal asterisk in a word or phrase, then escape the asterisk (precede it with the $\$ character).	
?	Question marks match only one character in the middle or end of your search term. For example, a search for <i>jo?n</i> finds items with the term <i>john</i> or <i>joan</i> but not <i>jon</i> or <i>johan</i> . You can't use a ? in a lookup search.	

When using wildcards, consider the following notes:

- The more focused your wildcard search, the faster the search results are returned, and the more likely the results will reflect your intention. For example, to search for all occurrences of the word prospect (or prospects, the plural form), it is more efficient to specify prospect* in the search string than to specify a less restrictive wildcard search (such as prosp*) that could return extraneous matches (such as prosperity).
- Tailor your searches to find all variations of a word. For example, to find property and properties, you would specify propert*.
- Punctuation is indexed. To find * or ? inside a phrase, you must enclose your search string in quotation marks and you must escape the special character. For example, "where are you\?" finds the phrase where are you?. The escape character (\) is required in order for this search to work correctly.

Operators

Combine multiple words with logic and grouping by using operators to form a more complex query. You can use the following special operators to focus your text search. Operator support is case-insensitive.

Operator	Description	
" "	Use quotation marks around search terms to find matches in the order you entered your search terms. A search for "monday meeting" finds items that contain monday meeting in that order.	
	To include the words "and," "or," and "and not" in your search results, surround those words in double quotes. Otherwise they're interpreted as the corresponding operators.	

Operator	Description
AND	Finds items that match all the search terms. For example, john AND smith finds items with both the word john and the word smith. Usually if an operator isn't specified, AND is the default operator. When searching articles, documents, and solutions, AND must be specified because OR is the default operator.
OR	Finds items with at least one of the search terms. For example, john OR smith finds items with either john or smith, or both words.
AND NOT	Finds items that do not contain the search term. For example, john AND NOT smith finds items that have the word john but not the word smith.
()	Use parentheses around search terms with logical operators to group search terms. For example, you can search for:
	 ("Bob" and "Jones") OR ("Sally" and "Smith") — searches for either Bob Jones or Sally Smith.
	• ("Bob") and ("Jones" OR "Thomas") and Sally Smith—searches for documents that contain Bob Jones and Sally Smith or Bob Thomas and Sally Smith.

SearchQuery Character Limits

If the SearchQuery string is longer than 10,000 characters, no result rows are returned. If SearchQuery is longer than 4,000 characters, any logical operators are removed. For example, the AND operator in a statement with a SearchQuery that's 4,001 characters will default to the OR operator, which could return more results than expected.

When you combine multiple operators in a search string, they're evaluated in this order:

- 1. Parentheses
- 2. AND and AND NOT (evaluated from right to left)
- **3.** OF

Reserved Characters

The following characters are reserved:

? & | ! { } [] () ^ ~ * : \ " ' + -

Reserved characters, if specified in a text search, must be escaped (preceded by the backslash \ character) to be properly interpreted. An error occurs if you do not precede reserved characters with a backslash. This is true even if the SearchQuery is enclosed in double quotes.

For example, to search for the following text:

{1+1}:2

insert a backslash before each reserved character:

\{1\+1\}\:2

Example FIND Clauses

Type of Search	Examples
Single term examples	FIND {MyProspect}
	<pre>FIND {mylogin@mycompany.com}</pre>
	FIND {FIND}
	FIND {IN}
	FIND {RETURNING}
	FIND {LIMIT}
Single phrase	FIND {John Smith}
Term OR Term	FIND {MyProspect OR MyCompany}
Term AND Term	FIND {MyProspect AND MyCompany}
Term AND Phrase	FIND {MyProspect AND "John Smith"}
Term OR Phrase	FIND {MyProspect OR "John Smith"}
Complex query using AND/OR	FIND {MyProspect AND "John Smith" OR MyCompany}
	<pre>FIND {MyProspect AND ("John Smith" OR MyCompany) }</pre>
Complex query using AND NOT	FIND {MyProspect AND NOT MyCompany}
Wildcard search	FIND {My*}
Escape sequences	FIND {Why not\?}
Invalid or incomplete phrase (will not succeed)	FIND {"John Smith}

FIND Clauses in Apex

The syntax of the FIND clause in Apex differs from the syntax of the FIND clause in the SOAP API and REST API:

• In Apex, the value of the FIND clause is demarcated with single quotes. For example:

```
FIND 'map*' IN ALL FIELDS RETURNING Account (Id, Name), Contact, Opportunity, Lead
```

• In the API, the value of the FIND clause is demarcated with braces. For example:

```
FIND {map*} IN ALL FIELDS RETURNING Account (Id, Name), Contact, Opportunity, Lead
```

The Lightning Platform Apex Code Developer's Guide has more information about using SOSL and SOQL with Apex.

FORMAT()

Use FORMAT with the FIND clause to apply localized formatting to standard and custom number, date, time, and currency fields.

When the FORMAT function is applied these fields reflect the appropriate format for the given user locale. The field format matches what appears in the Salesforce Classic user interface. For example, the date *December 28, 2015* can appear numerically as *2015-12-28, 28-12-2015, 28/12/2015, 12/28/2015*, or *28.12.2015*, depending on the org's locale setting.

The FORMAT function supports aliasing. In addition, aliasing is required when the query includes the same field multiple times. For example:

FIND {Acme} RETURNING Account(Id, LastModifiedDate, FORMAT(LastModifiedDate) FormattedDate)

You can also nest it with aggregate or convertCurrency () functions.

FIND {Acme} RETURNING Account (AnnualRevenue, FORMAT (convertCurrency (AnnualRevenue)) convertedCurrency)

IN SearchGroup

Specify which types of text fields to search for on a SOSL query by using the IN *SearchGroup* optional clause. For example, you can search for name, email, phone, sidebar, or all fields.

You can specify one of the following values (note that numeric fields are not searchable). If unspecified, the default behavior is to search all text fields in searchable objects.



Note: This clause doesn't apply to articles, documents, feed comments, feed items, files, products, and solutions. If these objects are specified in the RETURNING clause, the search is not limited to specific fields, and all fields are searched.

Valid SearchGroup Settings

Scope	Description
ALL FIELDS	Search all searchable fields. If the IN clause is unspecified, then this is the default setting.
EMAIL FIELDS	Search only email fields.
NAME FIELDS	Search only name fields.
	In addition to the standard Name field on most standard objects, these fields are also searched when using IN NAME FIELDS for these standard objects:
	Account: Website, Site, NameLocal
	Asset: SerialNumber
	Case: SuppliedName, SuppliedCompany, Subject
	 Contact: AssistantName, FirstNameLocal, LastNameLocal
	Event: Subject
	Lead: Company, CompanyLocal, FirstNameLocal, LastNameLocal
	Note: Title
	PermissionSet: Label
	Report: Description
	TagDefinition: NormName
	Task: Subject
	User: CommunityNickname

Scope	Description
	In custom objects, fields that are defined as "Name Field" are searched. In standard and custom objects, name fields have the nameField property set to true. (See the Field array of the fields parameter of the DescribeSObjectResult for more information.)
PHONE FIELDS	Search only phone number fields.
SIDEBAR FIELDS	Search for valid records as listed in the Sidebar drop-down list. Unlike search in the application, the asterisk (*) wildcard is not appended to the end of a search string.

While the IN clause is optional, it is recommended that you specify the search scope unless you need to search all fields. For example, if you're searching only for an email address, you should specify IN EMAIL FIELDS in order to design the most efficient search.

Example IN Clauses

Search Type	Example(s)
No search group	FIND {MyProspect}
ALL FIELDS	FIND {MyProspect} IN ALL FIELDS
EMAIL FIELDS	FIND {mylogin@mycompany.com} IN EMAIL FIELDS
NAME FIELDS	FIND {MyProspect} IN NAME FIELDS
PHONE FIELDS	FIND {MyProspect} IN PHONE FIELDS
SIDEBAR FIELDS	FIND {MyProspect} IN SIDEBAR FIELDS
Invalid search (will not succeed)	FIND {MyProspect} IN Accounts

LIMIT n

LIMIT is an optional clause that can be added to a SOSL query to specify the maximum number of rows that are returned in the text query, up to 2,000 results. If unspecified, the default is the maximum 2,000 results.

The default 2,000 results is the largest number of rows that can be returned for API version 28.0 and later. Previous versions return up to 200 results.

You can set limits on individual objects or on an entire query.

When you set a limit on the entire query, results are evenly distributed among the objects returned. For example, let's say you set an overall query limit of 20 and don't define any limits on individual objects. If 19 of the results are accounts and 35 are contacts, then only 10 accounts and 10 contacts are returned.

```
FIND {test} RETURNING Account(id), Contact LIMIT 20
```

Setting individual object limits allows you to prevent results from a single object using up the maximum query limit before other objects are returned. For example, if you issue the following query, at most 20 account records can be returned, and the remaining number of records can be contacts.

FIND {test} RETURNING Account(id LIMIT 20), Contact LIMIT 100

If you specify a limit of 0, no records are returned for that object.

OFFSET *n*

When expecting many records in a query's results, you can display the results in multiple pages by using the OFFSET clause in a SOSL query. For example, you can use OFFSET to display records 51 to 75 and then jump to displaying records 301 to 350. Using OFFSET is an efficient way to handle large results sets.

Use the optional OFFSET to specify the starting row offset into the result set returned by your query. Because the offset calculation is done on the server and only the result subset is returned, using OFFSET is more efficient than retrieving the full result set and then filtering the results locally. OFFSET can be used only when querying a single object. OFFSET must be the last clause specified in a query. OFFSET is available in API version 30.0 and later.

```
FIND {conditionExpression} RETURNING objectType(fieldList ORDER BY fieldOrderByList LIMIT number_of_rows_to_return OFFSET number_of_rows_to_skip)
```

As an example, if a query normally returned 50 rows, you could use OFFSET 10 in your query to skip the first 10 rows:

```
FIND {test} RETURNING Account (id LIMIT 10 OFFSET 10)
```

The result set for the preceding example would be a subset of the full result set, returning rows 11 through 20 of the full set.

Considerations When Using OFFSET

Consider these points when using OFFSET in your queries:

- The maximum offset is 2,000 rows. Requesting an offset greater than 2,000 will result in a System. SearchException:

 SOSI, offset should be between 0 to 2000 error
- We recommend using a LIMIT clause in combination with OFFSET if you need to retrieve subsequent subsets of the same result set. For example, you could retrieve the first 100 rows of a query using the following:

```
FIND {test} RETURNING Account(Name, Id ORDER BY Name LIMIT 100)
```

You could then retrieve the next 100 rows, 101 through 200, using the following query:

```
FIND {test} RETURNING Account(Name, Id ORDER BY Name LIMIT 100 OFFSET 100)
```

- When using OFFSET, only the first batch of records will be returned for a given query. If you want to retrieve the next batch, you'll need to re-execute the query with a higher offset value.
- Consecutive SOSL requests for the same search term but with a different OFFSET aren't guaranteed to return a different subset of the same data if the data being searched has been updated since the previous request.
- The Offset clause is allowed in SOSL used in SOAP API, REST API, and Apex.

ORDER BY Clause

You can specify the order in which search results are returned from a SOSL query using the ORDER BY clause. You can also use the clause to display empty records at the beginning or end of the results.

Use one or more ORDER BY clauses in a SOSL statement.

Syntax

|--|--|

Syntax	Description
ASC or DESC	Orders the results in ascending or descending order. The default is ascending. You can have more than one ORDER BY clause.
NULLS [first last]	Orders null records at the beginning (NULLS FIRST) or end (NULLS LAST) of the results. By default, null values are sorted first.

Examples

This example orders the account names in ascending ID order.

```
FIND {MyName} RETURNING Account (Name, Id ORDER BY Id)
```

This example, which shows multiple ORDER BY clauses, orders contacts in ascending order by name and by account description.

```
FIND {MyContactName} RETURNING Contact(Name, Id ORDER BY Name), Account(Description, Id ORDER BY Description)
```

This search returns account records in descending alphabetical order by name, with accounts that have null names appearing last.

```
FIND {MyAccountName} IN NAME FIELDS RETURNING Account(Name, Id ORDER BY Name DESC NULLS last)
```

This search returns custom objects that contain "San Francisco" in any field and have geolocation or address fields with locations that are within 500 miles of the latitude and longitude coordinates 37 and 122, respectively. The results are sorted in descending order by the locations' distance from the coordinates.

```
FIND {San Francisco} RETURNING My_Custom_Object__c (Name, Id WHERE
DISTANCE(My_Location_Field__c,GEOLOCATION(37,122),'mi') < 500 ORDER BY
DISTANCE(My_Location_Field__c,GEOLOCATION(37,122),'mi') DESC)</pre>
```

RETURNING FieldSpec

RETURNING is an optional clause that can be added to a SOSL query to specify the information to be returned in the text search result.

If unspecified, then the default behavior is to return the IDs of all objects that are searchable in advanced search as well as custom objects (even if they don't have a custom tab), up to the maximum specified in the LIMIT *n* clause or 2,000 (API version 28.0 and later), whichever is smaller. In the results, objects are listed in the order specified in the clause. For information on searchable fields in advanced and global search, see "Searchable Objects and Fields" in the Salesforce Help. API version 27.0 and earlier support a maximum of 200 results.



Note: External objects, articles, documents, feed comments, feed items, files, products, and solutions must be specified explicitly in a RETURNING clause to be returned in search results. For example:

FIND {MyProspect} RETURNING MySampleExternalObject, KnowledgeArticleVersion, Document, FeedComment, FeedItem, ContentVersion, Product2, Solution

Use the RETURNING clause to restrict the results data that is returned from the search () call. For information on IDs, see ID Field Type.

Syntax

In the following syntax statement, square brackets [] represent optional elements that can be omitted. A comma indicates that the indicated segment can appear more than one time.

```
RETURNING ObjectTypeName
[(FieldList [WHERE conditionExpression] [USING Listview=listview name] [ORDER BY Clause]
[LIMIT n] [OFFSET n])]
[, ObjectTypeName [(FieldList [WHERE conditionExpression] [ORDER BY Clause] [LIMIT n]
[OFFSET n])]]
```

RETURNING can contain the following elements:

Name	Description
ObjectTypeName	Object to return. If specified, then the search () call returns the IDs of all found objects matching the specified object. Must be a valid sObject type. You can specify multiple objects, separated by commas. If you specify more than one <code>ObjectTypeName</code> , each object must be distinct; you can't repeat an <code>ObjectTypeName</code> within a single RETURNING clause. Objects not specified in the RETURNING clause are not returned by the search () call.
FieldList	Optional list of one or more fields to return for a given object, separated by commas. If you specify one or more fields, the fields are returned for all found objects.
USING ListView=	Optional clause used to search within a single given object's list view. Only one list view can be specified. Only the first 2,000 records of the list view are searched, according to the sort order the user has set for the list view. ListView=Recent searches for the most recently accessed items viewed or referenced by the current user.
WHERE conditionExpression	Optional description of how search results for the given object should be filtered, based on individual field values. If unspecified, the search retrieves all the rows in the object that are visible to the user.
	Note that if you want to specify a WHERE clause, you must include a <code>FieldList</code> with at least one specified field. For example, this is not legal syntax:
	RETURNING Account (WHERE name like 'test')
	But this is:
	RETURNING Account (Name, Industry WHERE Name like 'test')
	See conditionExpression for more information.
ORDER BY Clause	Optional description of how to order the returned result, including ascending and descending order, and how nulls are ordered. You can supply more than one ORDER BY clause.
	Note that if you want to specify an ORDER BY clause, you must include a <code>FieldList</code> with at least one specified field. For example, this is not legal syntax:
	RETURNING Account (ORDER BY id)

Description
But this is:
RETURNING Account (Name, Industry ORDER BY Name)
Optional clause that sets the maximum number of records returned for the given object. If unspecified, all matching records are returned, up to the limit set for the query as a whole.
Note that if you want to specify a LIMIT clause, you must include a <i>FieldList</i> with at least one specified field. For example, this is not legal syntax:
RETURNING Account (LIMIT 10)
But this is:
RETURNING Account (Name, Industry LIMIT 10)
Optional clause used to specify the starting row offset into the result set returned by your query. OFFSET can be used only when querying a single object. OFFSET must be the last clause specified in a query.
Note that if you want to specify an OFFSET clause, you must include a <code>FieldList</code> with at least one specified field. For example, this is not legal syntax:
RETURNING Account (OFFSET 25)
But this is:
RETURNING Account (Name, Industry OFFSET 25)



Note: The RETURNING clause affects whether external objects are searched. For other objects, the RETURNING clause affects what data is returned, not what data is searched. The IN clause affects what data is searched.

Example RETURNING Clauses

Search Type	Example(s)
No Field Spec	FIND {MyProspect}
One sObject, no fields	FIND {MyProspect} RETURNING Contact
Multiple sObject objects, no fields	FIND {MyProspect} RETURNING Contact, Lead
One sObject, one or more fields	FIND {MyProspect} RETURNING Account(Name)
	FIND {MyProspect} RETURNING Contact(FirstName, LastName)
Custom sObject	FIND {MyProspect} RETURNING CustomObject_c
	FIND {MyProspect} RETURNING CustomObject_c(CustomField_c)
Multiple sObject objects, one or more fields, limits	<pre>FIND {MyProspect} RETURNING Contact(FirstName, LastName LIMIT 10), Account(Name, Industry)</pre>

Search Type	Example(s)
Multiple sObject objects, mixed number of fields	<pre>FIND {MyProspect} RETURNING Contact(FirstName, LastName), Account, Lead(FirstName)</pre>
Unsearchable sObject objects	FIND {MyProspect} RETURNING RecordType
	FIND {MyProspect} RETURNING Pricebook
Invalid sObject objects	FIND {MyProspect} RETURNING FooBar
Invalid sObject field	FIND {MyProspect} RETURNING Contact(fooBar)
Single object limit	<pre>FIND {MyProspect} RETURNING Contact(FirstName, LastName LIMIT 10)</pre>
Multiple object limits and a query limit	FIND {MyProspect} RETURNING Contact(FirstName, LastName LIMIT 20), Account(Name, Industry LIMIT 10), Opportunity LIMIT 50
Single object offset	<pre>FIND {MyProspect} RETURNING Contact(FirstName, LastName OFFSET 10)</pre>
List view	FIND {MyAcccount} IN ALL FIELDS RETURNING Account(Id, Name USING ListView=ListViewName)



Note: Apex requires that you surround SOQL and SOSL statements with square brackets to use them on the fly. You can use Apex script variables and expressions when preceded by a colon (:).

toLabel(fields)

Use to Label (fields) to translate SOSL query results into the user's language.

A client application can have results from a query returned that are translated into the user's language, using tolabel():

```
toLabel(object.field)
```

For example:

```
FIND {Joe} RETURNING Lead(company, toLabel(Recordtype.Name))
```

This query returns lead records with the record type name translated into the language for the user who issued the query.



Note: You can't filter on the translated name value from a record type. Always filter on the master value or the ID of the object for record types.

You can use to Label () to filter records using a translated picklist value. For example:

```
FIND {test} RETURNING Lead(company, toLabel(Status) WHERE toLabel(Status) = 'le Draft')
```

Lead records are returned where the picklist value for Status is 'le Draft.' The comparison is made against the value for the user's language. If no translation is available for the user's language for the specified picklist, the comparison is made against the master values.



Note: The tolabel() method cannot be used with the ORDER BY *Clause*. Salesforce always uses the order defined in the picklist, just like reports.

The toLabel function supports aliasing. In addition, aliasing is required when the query includes the same field multiple times. For example:

FIND {Joe} RETURNING Lead(company, toLabel(Recordtype.Name) AliasName)

Update an Article's Keyword Tracking with SOSL

Track keywords that are used in Salesforce Knowledge article searches with the UPDATE TRACKING optional clause on a SOSL query. You can use the language attribute to search by locale.

The UPDATE TRACKING clause is used to report on Salesforce Knowledge article searches and views. It allows developers to track the keywords used in Salesforce Knowledge article searches. Also, the language attribute can be used to search by a specific language (locale). However, only one language can be specified in a single query. Make a separate query for each language that you want. Use the Java format, which uses the underscore (for example, fr_FR, jp_JP, and so on), to supply locales. Search the Web for "java locale codes" to get a list of supported locales.

You can use this syntax to track a keyword used in Salesforce Knowledge article search:

```
FIND {Keyword}
RETURNING KnowledgeArticleVersion (Title WHERE PublishStatus="Online" and language="en_US")
UPDATE TRACKING
```

Update an Article's Viewstat with SOSL

Determine how many hits a Salesforce Knowledge article has had by using the UPDATE VIEWSTAT optional clause on a SOSL query. You can use the language attribute to search by locale.

The optional UPDATE VIEWSTAT clause is used to report on Salesforce Knowledge article searches and views. It allows developers to update an article's view statistics. Also, the language attribute can be used to search by a specific language (locale). However, only one language can be specified in a single query. Make a separate query for each language that you want. Use the Java format, which uses the underscore (for example, fr_FR, jp_JP, and so on), to supply locales. Search the Web for "java locale codes" to get a list of supported locales.

You can use this syntax to increase the view count for every article you have access to online in US English:

```
FIND {Title}

RETURNING FAQ__kav (Title WHERE PublishStatus="Online" and language="en_US" and KnowledgeArticleVersion = 'ka230000000PCiy')

UPDATE VIEWSTAT
```

USING Listview=

Optional clause used to search within a single given object's list view. Only one list view can be specified. Only the first 2,000 records of the list view are searched, according to the sort order the user has set for the list view. The clause can be used in API version 41 or later.

Example: The following SOSL statement searches through the MVP Customers list view for the Account object for Acme.

FIND {Acme} IN ALL FIELDS RETURNING Account(Id, Name USING ListView=MVPCustomers)

Example: The following SOSL statement searches through the recently viewed list view for the Account object for Acme.

FIND {Acme} IN ALL FIELDS RETURNING Account (Name USING LISTVIEW = Recent ORDER BY Name ASC NULLS FIRST, id ASC NULLS FIRST LIMIT 51)

Supported APIs

The clause in SOSL is supported in SOAP API, REST API, and Apex.

WHERE conditionExpression

By default, a SOSL query on an object retrieves all rows that are visible to the user, including archived rows. To limit the search, you can filter the search result by specific field values.

conditionExpression

The condition Expression of the WHERE clause uses the following syntax:

```
fieldExpression [logicalOperator fieldExpression2 ... ]
```

You can add multiple field expressions to a condition expression by using logical operators.

The condition expressions in SOSL FIND statements appear in bold in these examples:

- FIND {test} RETURNING Account (id WHERE createddate = THIS FISCAL QUARTER)
- FIND {test} RETURNING Account (id WHERE cf_c includes('AAA'))

You can use parentheses to define the order in which *fieldExpressions* are evaluated. For example, the following expression is true if fieldExpression1 is true and either fieldExpression2 or fieldExpression3 are true:

```
fieldExpression1 AND (fieldExpression2 OR fieldExpression3)
```

However, the following expression is true if either fieldExpression3 is true or both fieldExpression1 and fieldExpression2 are true.

```
(fieldExpression1 AND fieldExpression2) OR fieldExpression3
```

Client applications must specify parentheses when nesting operators. However, multiple operators of the same type do not need to be nested.

fieldExpression

A *fieldExpression* uses the following syntax:

```
fieldName comparisonOperator value
```

where:

Syntax	Description	
fieldName	The name of a field in the specified object. Use of single or double quotes around the name will result in an error. You must have at least read-level permissions to the field. It can be any field except a long text area field, encrypted data field, or base64-encoded field. It does not need to be a field in the fieldList.	
comparisonOperator	Case-insensitive operators that compare values.	
value	A value used to compare with the value in <code>fieldName</code> . You must supply a value whose data type matches the field type of the specified field. You must supply a native value—other field names or calculations are not permitted. If quotes are required (for example, they are not for dates and numbers), use single quotes. Double quotes result in an error.	

Comparison Operators

The following table lists the *comparisonOperator* values that are used in *fieldExpression* syntax. Comparisons on strings are case-insensitive.

Operator	Name	Description
=	Equals	Expression is true if the value in the specified <code>fieldName</code> equals the specified <code>value</code> in the expression. String comparisons using the equals operator are case-sensitive for unique case-sensitive fields and case-insensitive for all other fields.
!=	Not equals	Expression is true if the value in the specified fieldName does not equal the specified value.
<	Less than	Expression is true if the value in the specified fieldName is less than the specified value.
<=	Less or equal	Expression is true if the value in the specified $fieldName$ is less than, or equals, the specified $value$.
>	Greater than	Expression is true if the value in the specified $fieldName$ is greater than the specified $value$.
>=	Greater or equal	Expression is true if the value in the specified fieldName is greater than or equal to the specified value.
LIKE	Like	Expression is true if the value in the specified <code>fieldName</code> matches the characters of the text string in the specified <code>value</code> . The <code>LIKE</code> operator in SOQL and SOSL is similar to the <code>LIKE</code> operator in SQL; it provides a mechanism for matching partial text strings and includes support for wildcards.
		• The % and _ wildcards are supported for the LIKE operator.
		• The % wildcard matches zero or more characters.
		The _ wildcard matches exactly one character.
		 The text string in the specified value must be enclosed in single quotes.
		 The LIKE operator is supported for string fields only.
		 The LIKE operator performs a case-insensitive match, unlike the case-sensitive matching in SQL.

Operator Name		Description	
		 The LIKE operator in SOQL and SOSL supports escaping of special characters % or 	
		• Don't use the backslash character in a search except to escape a special character.	
		For example, the following query matches Appleton, Apple, and Appl, but not Bappl:	
		SELECT AccountId, FirstName, lastname FROM Contact WHERE lastname LIKE 'appl%'	
IN IN		If the value equals any one of the specified values in a WHERE clause. For example:	
		SELECT Name FROM Account WHERE BillingState IN ('California', 'New York')	
		The values for IN must be in parentheses. String values must be surrounded by single quotes.	
		IN and NOT IN can also be used for semi-joins and anti-joins when querying on ID (primary key) or reference (foreign key) fields.	
NOT IN NOT IN	NOT IN	If the value does not equal any of the specified values in a WHERE clause. For example:	
		SELECT Name FROM Account WHERE BillingState NOT IN ('California', 'New York')	
		The values for NOT IN must be in parentheses, and string values must be surrounded by single quotes.	
		There is also a logical operator NOT, which is unrelated to this comparison operator.	
INCLUDES EXCLUDES		Applies only to multi-select picklists.	

Logical Operators

The following table lists the logical operator values that are used in fieldExpression syntax:

Operator	Syntax	Description
AND	fieldExpressionX AND fieldExpressionY	true if both fieldExpressionX and fieldExpressionY are true.
OR	fieldExpressionX OR	true if either fieldExpressionX or fieldExpressionY is true.
fieldExpressionY	fieldExpressionY	Relationship queries with foreign key values in an OR clause behave differently depending on the version of the API. In a WHERE clause that uses OR, if the foreign key value in a record is null, the record is returned in version 13.0 and later, but not returned in versions before 13.0.
	<pre>SELECT Id FROM Contact WHERE LastName = 'foo' or Account.Name = 'bar'</pre>	

Operator	Syntax	Description
		The contact with no parent account has a last name that meets the criteria, so it is returned in version 13.0 and later.
NOT	not fieldExpressionX	true if fieldExpressionX is false.
		There is also a comparison operator ${\tt NOT}$ ${\tt IN}$, which is different from this logical operator.

Quoted String Escape Sequences

You can use the following escape sequences with SOSL:

Sequence	Meaning
\n or \N	New line
\r or \R	Carriage return
\t or \T	Tab
\b or \B	Bell
\f or \F	Form feed
\"	One double-quote character
\'	One single-quote character
\\	Backslash
LIKE expression only: _	Matches a single underscore character (_)
LIKE expression only:\%	Matches a single percent sign character (%)

If you use a backslash character in any other context, an error occurs.

Example WHERE Clauses

Example(s)

```
FIND {test}
    RETURNING Account (id WHERE createddate = THIS_FISCAL_QUARTER)
```

```
FIND {test}

RETURNING Account (id WHERE cf__c includes('AAA'))
```

Example(s)

```
FIND {test}
    RETURNING Account (id), User(Field1, Field2 WHERE Field1 = 'test' order by id ASC,
Name DESC)
```

```
FIND {test} IN ALL FIELDS
    RETURNING Contact(Salutation, FirstName, LastName, AccountId WHERE Name = 'test'),
        User(FirstName, LastName),
        Account(id WHERE BillingState IN ('California', 'New York'))
```

```
FIND {test}
    RETURNING Account (id WHERE (Name = 'New Account')
    or (Id = '001z00000008Vq7'
    and Name = 'Account Insert Test')
    or (NumberOfEmployees < 100 or NumberOfEmployees = null)
    ORDER BY NumberOfEmployees)</pre>
```

To search for a Salesforce Knowledge article by ID:

```
FIND {tourism}
    RETURNING KnowledgeArticleVersion (Id, Title WHERE id = 'ka0D0000000025eIAA')
```

To search for multiple Salesforce Knowledge articles by ID:

```
FIND {tourism}
    RETURNING KnowledgeArticleVersion
    (Id, Title WHERE id IN ('ka0D0000000025eIAA', 'ka0D000000002HCIAY'))
```

To search for "San Francisco" in all fields of all My_Custom_Object__c objects that have a geolocation or address location within 500 miles of the latitude and longitude coordinates 37 and 122, respectively:

```
FIND {San Francisco}
    RETURNING My_Custom_Object__c (Id
        WHERE DISTANCE(My_Location_Field__c,GEOLOCATION(37,122),'mi') < 100)</pre>
```

WITH DATA CATEGORY DataCategorySpec

WITH DATA CATEGORY is an optional clause that can be added to a SOSL query to filter all search results that are associated with one or more data categories and are visible to users. This clause is used in searches of Salesforce Knowledge articles and questions.

The WITH DATA CATEGORY clause can be used in API version 18.0 or later.

Syntax

The WITH DATA CATEGORY syntax is:

```
WITH DATA CATEGORY DataCategorySpec [logicalOperator DataCategorySpec2 ... ]
```

Where DataCategorySpec consists of a groupName, Operator, and category.

Name	Description	
groupName	The name of the data category group to filter. For information on category groups, see "Create and Modify Category Groups" in the Salesforce Help.	
Operator	Use one of the following operators:	
	 AT—Queries the specified data category. 	
	 ABOVE—Queries the specified data category and all of its parent categories. 	
	 BELOW—Queries the specified data category and all of its subcategories. 	
	 ABOVE_OR_BELOW—Queries the specified data category, all of its parent categories, and all of its subcategories. 	
category	The name of the category to filter. To include multiple data categories, enclose them in parentheses, separated by commas. For information on categories, see "Add Data Categories to Category Groups" in the Salesforce Help.	

You can add multiple data category specifiers by using the logical operator AND. Other operators, such as OR and AND NOT, are not supported.

A SOSL statement using the WITH DATA CATEGORY clause must also include a RETURNING **ObjectTypeName** clause, with a WHERE clause that filters on the PublishStatus field.

In the RETURNING clause, specify one of the following for <code>ObjectTypeName</code>:

- To search a specific article type, use the article type name with the suffix ___kav
- To search all article types, use KnowledgeArticleVersion
- To search questions, use Question

For information on article types, see "Knowledge Article Types" in the Salesforce Help.

The WHERE clause must use one of the following publish statuses:

- WHERE PublishStatus='online' for published articles
- WHERE PublishStatus='archived' for archived articles
- WHERE PublishStatus='draft' for draft articles

Examples

Search Type	Example
Search all published (online)Salesforce Knowledge articles with a category from one category group.	FIND {tourism} RETURNING KnowledgeArticleVersion (Id, Title WHERE PublishStatus='online') WITH DATA CATEGORY Locationc AT Americac
Search online FAQ articles with categories from two category groups.	<pre>FIND {tourism} RETURNING FAQ_kav (Id, Title WHERE PublishStatus='online')</pre>

Search Type	Example	
	WITH DATA CATEGORY Geographyc ABOVE Francec AND Productc AT mobile_phonesc	
Search archived FAQ articles from one category group.	FIND {tourism} RETURNING FAQ_kav (Id, Title WHERE PublishStatus='archived') WITH DATA CATEGORY Geography_c AT Iceland_c	
Search all draft Salesforce Knowledge articles from one category group.	FIND {tourism} RETURNING KnowledgeArticleVersion (Id, Title WHERE PublishStatus='draft') WITH DATA CATEGORY Geography_c BELOW Europe_c	

For information on the WITH DATA CATEGORY clause, see the WITH DATA CATEGORY filteringExpression.



🊺 Tip: You can also search for articles by ID, without using the WITH DATA CATEGORY clause. For more information, see Example WHERE Clauses.

WITH DivisionFilter

WITH DivisionFilter is an optional clause that can be added to a SOSL query to filter all search results based on the division field. It pre-filters all records based on the division before applying other filters. You can also specify a division by its name rather than by its ID.

For example:

```
FIND {test} RETURNING Account (id where name like '%test%'),
                     Contact (id where name like '%test%')
            WITH DIVISION = 'Global'
```



Note:

- Users can perform searches based on division regardless of whether they have the "Affected by Divisions" permission enabled.
- All searches within a specific division also include the global division. For example, if you search within a division called Western Division, your results include records found in both the Western Division and the global division.

WITH HIGHLIGHT

WITH HIGHLIGHT is an optional clause that can be added to a SOSL query for business account, campaign, contact, custom object, lead, opportunity, quote, and user searches. It highlights the terms matching the search query in search results, making it easier to identify relevant content. The WITH HIGHLIGHT clause can be used in API version 39.0 or later. Using WITH HIGHLIGHT for custom fields and objects can be used in API version 40.0 and later.

Highlighted search terms are generated from the following standard and custom field types.

- Auto number
- Email
- Text

- Text Area
- Text Area (Long)

Highlighted search terms are not generated from the following field types.

- Checkbox
- Compound fields
- Currency
- Date
- Date/Time
- File
- Formula
- Lookup Relationship
- Number
- Percent
- Phone
- Picklist
- Picklist (Multi-Select)
- Text Area (Rich)
- URL
- Example: The following SOSL statement returns search results with the search term salesforce highlighted.

```
FIND {salesforce} IN ALL FIELDS RETURNING Account(Name, Description) WITH HIGHLIGHT
```

The matching terms are highlighted with <mark> tags. If the original search term doesn't yield any results because of incorrect spelling, the corrected spelling of the search term is highlighted in the results.

Example:

```
"searchRecords" : [ {
    "attributes" : {
        "type" : "Account",
        "url" : "/services/data/v39.0/sobjects/Account/001xx000003DpxkAAC"
    },
    "Name" : "salesforce",
    "Description" : "Salesforce.com",
    "highlight.Description" : "<mark>salesforce</mark>.com",
    "highlight.Name" :
        "<mark>salesforce</mark>"
} ]
```

Example: The following statement returns search results with the search term salesforce west highlighted for the custom field BuildingDescription of the custom object Building.

```
FIND {Salesforce West} IN ALL FIELDS RETURNING Building_c(Name, BuildingDescription_c) WITH HIGHLIGHT
```

Usage

Search terms that contain a wildcard are not highlighted.

Other objects that are included in searches that contain WITH HIGHLIGHT don't return highlighted search terms.

A maximum of 25 records per entity per SOSL query are highlighted.

Supported APIs

The WITH HIGHLIGHT clause in SOSL is supported in SOAP API and REST API.

WITH METADATA

Specifies if metadata is returned in the response. Optional clause.

No metadata is returned by default. To include metadata in the response, use the LABELS value, which returns the display label for the fields returned in search results. For example:

```
FIND {Acme} RETURNING Account(Id, Name) WITH METADATA='LABELS'
```

WITH NETWORK Network IdSpec

You can search for community users and feeds by using the WITH NETWORK optional clause on a SOSL query. When you're filtering search results by community, each community is represented by a community ID (NetworkId).

You can use the following syntax.

- WITH NETWORK IN ('NetworkId1', 'NetworkId2', ...) supports filtering by one or more communities.
- WITH NETWORK = 'NetworkId' supports filtering by a single community only.

For objects other than users and feeds, search results include matches across all communities and internal company data, even if you use network filtering in your query.

- You can run searches against multiple objects in the same community.
- You can't run scoped and unscoped searches in the same query. For example, you can't search users from a community along with accounts from the entire organization.

To filter search results for *groups* or *topics* by community, use the WHERE clause with a Networkld value. If you want to search for an internal community, use an all zero value for Networkld.

Example WITH NETWORK Network IdSpec Clauses

To search multiple communities for users and feed items containing the string "test" and to sort feed items from the newest to the oldest:

```
FIND {test} RETURNING User (id),

FeedItem (id, ParentId WHERE CreatedDate =

THIS_YEAR Order by CreatedDate DESC)

WITH NETWORK IN ('NetworkId1', 'NetworkId2', 'NetworkId3')
```

To search the Networkld community for users and feed items containing the string "test" and to sort feed items from the newest to the oldest:

```
FIND {test} RETURNING User (id),

FeedItem (id, ParentId WHERE CreatedDate =

THIS_YEAR Order by CreatedDate DESC)

WITH NETWORK = 'NetworkId'
```

To search in an internal community for users and feed items containing the string "test" and to sort feed items from newest to oldest:

```
FIND {test} RETURNING User (id),

FeedItem (id, ParentId WHERE CreatedDate =

THIS_YEAR Order by CreatedDate DESC)

WITH NETWORK = '00000000000000'
```

WITH PricebookId

Filters product search results by a single price book ID.

Only applicable for the Product2 object. The price book ID must be associated with the product that you're searching for. For example:

```
Find {laptop} RETURNING Product2 WITH PricebookId = '01sxx000002MffAAE'
```

WITH SNIPPET

WITH SNIPPET is an optional clause that can be added to a SOSL query for article, case, feed, and idea searches. On the search results page, excerpts below article titles show terms matching the search query highlighted within the context of surrounding text. Snippets make it easier for users to identify the content they're looking for.



Note: To generate search results with highlighted matches but not snippets, use WITH HIGHLIGHT.

Search snippets and highlights are generated from the following field types.

- Email
- Text
- Text Area
- Text Area (Long)
- Text Area (Rich)

Search snippets and highlights are not generated from the following field types.

- Checkbox
- Currency
- Date
- Date/Time
- File
- Formula
- Lookup Relationship
- Number
- Percent

- Phone
- Picklist
- Picklist (Multi-Select)
- URL
- Example: The following SOSL statement returns snippets for articles that match the search term San Francisco.

The matching terms are highlighted with <mark> tags within the context of the snippet results. Stemmed forms of the term and any synonyms defined are also highlighted.

Example:

```
[ {
 "attributes" : {
   "type" : "KnowledgeArticleVersion",
   "url": "/services/data/v32.0/sobjects/KnowledgeArticleVersion/kaKD000000001MAA"
 },
 "Id" : "kaKD0000000001MAA"
 "Title" : "San Francisco"
 "Summary" : "City and County of San Francisco"
 "snippet.text" : "<mark>San</mark> <mark>Francisco</mark>, officially the City and
County of <mark>San</mark> <mark>Francisco</mark> is the... City and County of
<mark>San</mark> <mark>Fran</mark>"
 "highlight.Title" : "<mark>San</mark> <mark>Francisco</mark>"
}, {
  "attributes" : {
    "type" : "KnowledgeArticleVersion",
    "url": "/services/data/v32.0/sobjects/KnowledgeArticleVersion/kaBD0000000007DMAQ"
  "Id" : "kaBD0000000007DMAQ",
  "Title" : "San Francisco Bay Area",
  "Summary": "Nine county metropolitan area",
  "snippet.text" : "The <mark>SF</mark> Bay Area, commonly known as the Bay Area, is
a populated region that"
  "highlight.Title" : "<mark>San</mark> <mark>Francisco</mark> Bay Area"
  "attributes" : {
    "type" : "KnowledgeArticleVersion",
    "url": "/services/data/v32.0/sobjects/KnowledgeArticleVersion/ka3D0000000042OIAQ"
  },
  "Id" : "ka3D0000000420IAQ",
  "Title" : "California",
  "Summary" : "State of California",
  "snippet.text": "(Greater Los Angeles area and <mark>San</mark>
<mark>Francisco</mark> Bay Area, respectively), and eight of the nation's 50 most"
 } ]
```



Note: In this example, "SF" (as a synonym defined for "San Francisco") and "San Fran" (as a stemmed form of "San Francisco") are also highlighted in the results as matching terms.

Usage

For SOSL statements using the WITH SNIPPET clause, we recommend using a RETURNING ObjectTypeName clause, with a WHERE clause that filters on the PublishStatus field.

In the RETURNING clause, specify one of the following for ObjectTypeName:

- To search a specific article type, use the article type name with the suffix <u>kav</u>.
- To search all article types, use KnowledgeArticleVersion.
- To search case, case comment, feed, feed comment, idea, and idea comment types, use Case, CaseComment, FeedItem, FeedComment, Idea, and IdeaComment. For example:

```
FIND {San Francisco} IN ALL FIELDS RETURNING FeedItem, FeedComment WITH SNIPPET
(target length=120)
```

Other objects that are included in searches that contain WITH SNIPPET don't return snippets.

Snippets aren't displayed for search terms that contain a wildcard, when the search doesn't return any articles, or if the user doesn't have access to the field that contains the snippet. Even if you add the WITH SNIPPET clause, searches that don't return snippets don't return snippets.

Snippets are only displayed when 20 or fewer results are returned on a page.



🁔 Tip: Use the LIMIT or OFFSET clause to return only 20 results at a time.

Escaped HTML Tags

When matching terms within HTML tags are returned in a snippet, the HTML tags are escaped and the matching terms are highlighted in the results.



Example: A search for salesforce returns an article with the text "For more information, visit salesforce.com". The original hyperlink tags from the article are escaped (encoded) and "salesforce" is highlighted in the snippet result.

```
For more information, visit <a
href='http://salesforce.com'>salesforce.com</a&gt;
```

Target Snippet Length

By default, each snippet displays up to approximately 300 characters, which is usually three lines of text in a standard browser window display. The number of characters displayed is the target length, within a statistically insignificant degree of variance.

Snippets consist of one or more fragments of text that contain the matching terms. If the returned snippet includes multiple text fragments (for example, for matches within multiple fields), the target length is the maximum total length of all the returned fragments.

To specify an alternate target length, add the optional target length parameter to the WITH SNIPPET clause. You can specify a target length from 50 to 1,000 characters. When the target length is set to an invalid number, such as 0 or a negative number, the length defaults to 300.



Example: A target_length parameter of 120 characters is useful for displaying a snippet of approximately three lines of text in a standard mobile interface.

Supported APIs

The WITH SNIPPET clause can be used in API version 32.0 or later. The WITH SNIPPET clause in SOSL is supported in SOAP API, REST API, and Apex.

WITH SPELL CORRECTION

WITH SPELL_CORRECTION is an optional clause that can be added to a SOSL query. When set to true, spell correction is enabled for searches that support spell correction. When set to false, spell correction is not enabled. The default value is true. The WITH SPELL_CORRECTION clause can be used in API version 40 or later.



Example: The following SOSL statement disables spell correction on a search of accounts for the term San Francisco.

FIND {San Francisco} IN ALL FIELDS RETURNING Account WITH SPELL_CORRECTION = false

Supported APIs

The WITH SPELL CORRECTION clause in SOSL is supported in SOAP API, REST API, and Apex.

INDEX

A	F
Anti-join 15	FIND and Apex 93
Apex and SOSL 93 Async SOQL	R
Overview 79	Reserved characters in SOSL FIND 92
В	S
Big Objects	SearchQuery character limit 92
Querying 74	Semi-join 15
C	SOSL
C	Apex syntax 93
Characters reserved in SOSL FIND 92	SOSL character limit 92