
SOQL and SOSL Reference

Version 59.0, Winter '24



CONTENTS

Chapter 1: Introduction to SOQL and SOSL	1
Chapter 2: Salesforce Object Query Language (SOQL)	3
Typographical Conventions in This Document	5
Quoted String Escape Sequences	5
Reserved Characters	6
Alias Notation	7
SOQL SELECT Syntax	7
SELECT	10
TYPEOF	14
USING SCOPE	16
WHERE	17
WITH	34
GROUP BY	38
HAVING	45
ORDER BY	46
LIMIT	47
OFFSET	48
FOR VIEW and FOR REFERENCE	50
UPDATE	51
FOR UPDATE	51
SOQL SELECT Examples	51
SOQL SELECT Functions	53
Aggregate Functions	54
convertCurrency()	58
convertTimezone()	59
Date Functions	59
FORMAT ()	62
GROUPING(fieldName)	62
toLabel()	62
Relationship Queries	63
Understanding Relationship Names	64
Using Relationship Queries	65
Understanding Relationship Names, Custom Objects, and Custom Fields	66
Understanding Query Results	68
null Values in Lookup Relationships and Outer Joins	70
Identifying Parent and Child Relationships	70
Understanding Relationship Fields and Polymorphic Fields	72
Understanding Relationship Query Limitations	77

Contents

Using Relationship Queries with History Objects	78
Using Relationship Queries with Data Category Selection Objects	78
Using Relationship Queries with the Partner WSDL	78
Change the Batch Size in Queries	78
SOQL Limits on Objects	79
SOQL with Big Objects	83
Syndication Feed SOQL and Mapping Syntax	84
Location-Based SOQL Queries	85
Chapter 3: Salesforce Object Search Language (SOSL)	89
Typographical Conventions in This Document	91
SOSL Limits on Search Results	91
SOSL Limits on External Object Search Results	93
SOSL Syntax	93
Example Text Searches	96
convertCurrency()	97
FIND {SearchQuery}	98
FORMAT()	102
IN SearchGroup	102
LIMIT n	103
OFFSET n	104
ORDER BY Clause	105
RETURNING FieldSpec	105
toLabel(fields)	108
Update an Article's Keyword Tracking with SOSL	109
Update an Article's Viewstat with SOSL	109
USING Listview=	110
WHERE	110
WITH DATA CATEGORY DataCategorySpec	114
WITH DivisionFilter	116
WITH HIGHLIGHT	116
WITH METADATA	118
WITH NETWORK NetworkIdSpec	118
WITH PricebookId	119
WITH SNIPPET	119
WITH SPELL_CORRECTION	122
Index	123

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.

 **Note:** SOSL doesn't support big objects.

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 many fields leads to many permutations, which can be difficult to tune.

For more information, see [Best Practices for Deployments with Large Data Volumes](#).

Sending Queries and Searches

Construct and execute queries and searches using one of several environments, including SOAP API, REST API, Apex, and more. For more information on which environments support queries and searches, see [Salesforce Object Query Language \(SOQL\)](#) on page 3 and [Salesforce Object Search Language \(SOSL\)](#) on page 89.

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](#)
- [SOQL SELECT Examples](#)
- [SOQL SELECT Functions](#)
- [Relationship Queries](#)
- [Change the Batch Size in Queries](#)
- [SOQL Limits on Objects](#)
- [Syndication Feed SOQL and Mapping Syntax](#)
- [Location-Based SOQL Queries](#)

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 query strings in several environments.

- Using the `queryString` parameter of a SOAP API `query()` call. See [query\(\)](#) in the *SOAP API Developer Guide*.
- Using the `q` parameter of a REST API query request. See [Query](#) in the *REST API Developer Guide*.
- Using Apex statements. See [SOQL and SOSL Queries](#) in the *Apex Developer Guide*.
- Using Visualforce controllers and getter methods. See [Controller Methods](#) in the *Visualforce Developer Guide*.
- Using the Salesforce CLI. See [force:data:soql:query](#) in the data Commands topic of the *Salesforce CLI Command Reference*.
- Using an extension for Visual Studio Code. See [Write SOQL Queries](#) in *Salesforce Extensions for Visual Studio Code*.

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 in your statements. 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.

 **Note:** With archived data and big objects, you can use only some SOQL features. For more information, see [SOQL with Big Objects](#) on page 83.

SEE ALSO:

[Salesforce Developer Limits and Allocations Quick Reference: SOQL and SOSL Limits for Search Queries](#)

Typographical Conventions in This Document

This SOQL reference uses custom typographical conventions.

Convention	Description
<code>SELECT Name FROM Account</code>	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.
<code>SELECT <i>fieldname</i> FROM <i>objectname</i></code>	Italics represent a variable or placeholder. You supply the actual value.
<code>{ }</code>	Curly braces group elements to remove ambiguity. For example, in the clause <code>UPDATE {TRACKING VIEWSTAT} [, ...]</code> , the curly braces indicate that the pipe shows a choice between <code>TRACKING</code> and <code>VIEWSTAT</code> after <code>UPDATE</code> , rather than a choice between <code>UPDATE TRACKING</code> and <code>VIEWSTAT</code> .
<code> </code>	The pipe character separates alternate elements. For example, in the clause <code>UPDATE {TRACKING VIEWSTAT} [, ...]</code> , the <code> </code> character indicates that you can use either <code>TRACKING</code> or <code>VIEWSTAT</code> after <code>UPDATE</code> .
<code>[]</code>	Square brackets indicate an optional element. For example, <code>[LIMIT rows]</code> means that you can specify zero or one <code>LIMIT</code> 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 <code>[ORDER BY fieldOrderByList [ASC DESC] [NULLS {FIRST LAST}]]</code> , <code>ASC</code> , <code>DESC</code> , or the <code>NULLS</code> clause cannot be used without the <code>ORDER BY</code> clause.
<code>[...] and [, ...]</code>	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 <code>UPDATE {TRACKING VIEWSTAT} [, ...]</code> , the <code>[, ...]</code> indicates that you can use <code>TRACKING</code> , <code>VIEWSTAT</code> , or both: <div data-bbox="641 1417 1445 1585" style="border: 1px solid #add8e6; padding: 5px; margin-top: 10px;"> <pre>UPDATE TRACKING</pre> <pre>UPDATE VIEWSTAT</pre> <pre>UPDATE TRACKING, VIEWSTAT</pre> </div>

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 (%)
\uXXXX	A Unicode character with XXXX as the code (for example, \u00e9 represents the é 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 backslash (\) 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, USING, 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 | FOR REFERENCE} ]
[UPDATE {TRACKING|VIEWSTAT} ]
[FOR UPDATE]
```

Syntax

Description

fieldList subquery

Specifies a list of one or more fields, separated by commas, that you want to retrieve from the specified *object*. The bold elements in the following examples are *fieldlist* values:

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

Syntax	Description
	<ul style="list-style-type: none"> • <code>SELECT Contact.Firstname, Contact.Account.Name FROM Contact</code> • <code>SELECT FIELDS(STANDARD) FROM Contact</code> <p>Use valid field names and include read-level permissions for each specified field. The <i>fieldList</i> defines the ordering of fields in the query results.</p> <p>If the query traverses a relationship, <i>fieldList</i> can include a subquery. For example:</p> <pre>SELECT Account.Name, (SELECT Contact.LastName FROM Account.Contacts) FROM Account</pre> <p>The <i>fieldlist</i> can also be an aggregate function, such as <code>COUNT ()</code> and <code>COUNT (<i>fieldName</i>)</code>, or be wrapped in the <code>toLabel()</code> function to translate returned results. See SELECT for more information.</p>
<i>typeOfField</i>	A polymorphic relationship field in <i>objectType</i> or a polymorphic field in a parent of <i>objectType</i> that can reference multiple object types. For example, the <code>who</code> relationship field of a Task can be a Contact or a Lead. <i>typeOfField</i> can't reference a relationship field that is also referenced in <i>fieldList</i> . See TYPEOF for more information.
<i>whenExpression</i>	A clause of the form <code>WHEN <i>whenObjectType</i> THEN <i>whenFieldList</i></code> . You can have one or more <i>whenExpression</i> clauses inside a <code>TYPEOF</code> expression. See TYPEOF for more information.
<i>elseExpression</i>	A clause of the form <code>ELSE <i>elseFieldList</i></code> . This clause is optional inside a <code>TYPEOF</code> expression. See TYPEOF for more information.
<i>objectType</i>	Specifies the type of object that you want to <code>query ()</code> . Provide a valid object, such as Account, and must have read-level permissions to that object.
<i>filterScope</i>	Available in API version 32.0 and later. Specifies the <i>filterScope</i> for limiting the results of the query.
<i>conditionExpression</i>	If <code>WHERE</code> is specified, determines which rows and values in the specified object (<i>objectType</i>) to filter against. If unspecified, the <code>query ()</code> retrieves all the rows in the object that are visible to the user.
<i>filteringExpression</i>	<p>If <code>WITH DATA CATEGORY</code> is specified, the <code>query ()</code> only returns matching records that are associated with the specified data categories and are visible to the user. If unspecified, the <code>query ()</code> returns the matching records that are visible to the user. The <code>WITH DATA CATEGORY</code> clause only filters objects of type:</p> <ul style="list-style-type: none"> • <code>Question</code>—to query questions. • <code>KnowledgeArticleVersion</code>—to query articles. <p>See WITH DATA CATEGORY</p> <p>If <code>WITH DATA CATEGORY</code> isn't specified, <i>filteringExpression</i> can be used to enforce field- and object-level permission or to specify user- or system-mode access in Apex code.</p>

Syntax	Description
	<p>SECURITY_ENFORCED</p> <p>Use the <code>WITH SECURITY_ENFORCED</code> clause 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 in the <i>Apex Developer Guide</i>.</p> <p>USER_MODE or SYSTEM_MODE</p> <p>Specify user mode or system mode access for Apex database operations. Apex code runs in system mode by default, which means that it runs with substantially elevated permissions over the user running the code. To enhance the security context of Apex, you can specify user mode access using <code>WITH USER_MODE</code>. Field-level security (FLS) and object permissions of the running user are respected and sharing rules are applied in user mode, unlike in system mode. See Enforce User Mode for Database Operations (Beta) in the <i>Apex Developer Guide</i>.</p>
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 <code>GROUP BY</code> clause is used with aggregate functions to summarize the data. This clause also lets you 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 <code>GROUP BY</code> clause, this conditional expression filters the records that the <code>GROUP BY</code> returns. See HAVING .
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: <div data-bbox="568 1213 1006 1308" data-label="Text"> <pre>SELECT Id, LastName, FirstName FROM Contact ORDER BY LastName, FirstName</pre> </div>

Implementation Tips for SELECT Statement

- **Statement Character Limit**—By default, SOQL statements can't exceed 100,000 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 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 100,000 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.

- **Statements Returning Many Results**—If a SELECT statement returns a large amount of data per record, SOQL automatically reduces the number of results. The number of results can also vary based on how a SOQL query is called and on the complexity of the query. When you search a list view, only the first 2,000 records in the list are searched. If the object contains formula fields, derived fields, or CLOB or BLOB fields, large volumes of data can also be returned. To retrieve all the pages of results, use one of these methods:
 - In SOQL, if you use OFFSET and LIMIT in your SOQL query, the number of records returned can be less than LIMIT. Check the number of returned results and adjust OFFSET as required. Don't increment OFFSET by LIMIT.
 - In SOAP API, use `queryMore()`.
 - In REST API, use the `nextRecordsUrl` returned by `/query` and `/queryAll`.
 - In Bulk API 2.0, use the `Sforce-Locator` response header that is returned in the job's results.
- **Localized Results**—SELECT statements can include the `toLabel()`, `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 in your statements. You can use Apex script variables and expressions when preceded by a colon (:).
- **Ordered Results**—The order of results isn't 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 can be different from the actual value.
- **Avoid Query Timeouts**—Consider using `WHERE` or `WITH` clauses to reduce the query operation time. In REST API, you can also try the `explain` parameter (beta) to get information on query processing. For more information, see [Get Feedback on Query Performance](#) in the *REST API Developer Guide*.

SEE ALSO:

[Salesforce Developer Limits and Allocations Quick Reference: SOQL and SOSL Limits for Search Queries](#)

SELECT

SOQL query syntax consists of a required `SELECT` statement that specifies the fields to query. The ***fieldList*** in the `SELECT` statement specifies the list of one or more fields, separated by commas, that you want to retrieve.

```
SELECT fieldList [subquery] [...]
```

Use the `FIELDS()` keyword in the ***fieldList*** to select groups of fields without knowing their names in advance. This keyword simplifies SELECT statements, avoids the need for multiple API calls, and provides a low-code method to explore the data in your org. This keyword is available in API version 51.0 and later.

Using a field list in a SOQL SELECT statement (for example, `SELECT Id, Name FROM Account`) can be complicated if you need to retrieve many fields. And if you don't know what fields an object has, you must first get a description of the object. Typically, you use a call to first get a description of the object, then parse the description to identify the fields. Then you construct a SOQL query that specifies the fields, and then make another call to submit the query.

To simplify this process, the `FIELDS()` keyword lets you select fields easily, without knowing their names in advance. The `FIELDS()` keyword eliminates the need for a round trip to the server to prepare a SOQL statement, simplifies query statements, and makes it much easier to explore the shape of your objects. It also helps avoid exceeding the character limits on SOQL queries and the URI length limit for REST calls.

You can include any of these in the field list:

- `FIELDS (ALL)` —select all the fields of an object.
- `FIELDS (CUSTOM)` —select all the custom fields of an object.
- `FIELDS (STANDARD)` —select all the standard fields of an object.

In each case, `FIELDS ()` respects field-level security so it only shows the fields that you have permission to access.

Usage

You can use `FIELDS ()` as the complete field list. For example:

- `SELECT FIELDS(ALL) FROM Account LIMIT 200`
- `SELECT FIELDS(CUSTOM) FROM Account LIMIT 200`
- `SELECT FIELDS(STANDARD) FROM Account`

You can also use `FIELDS ()` with other field names in the field list. For example:

- `SELECT Name, Id, FIELDS(CUSTOM) FROM Account LIMIT 200`
- `SELECT someCustomField__c, FIELDS(STANDARD) FROM Account`

The `FIELDS ()` keyword can also be used in subqueries. For example:

```
SELECT
  Account.Name,
  (SELECT FIELDS(ALL) FROM Account.Contacts LIMIT 200)
FROM Account
```

The API returns an error if the `SELECT` statement results in any duplicated field names. For example, this query:

```
SELECT Id, FIELDS(ALL) FROM User LIMIT 200
```

gives this error:

```
HTTP/1.1 400 Bad Request
[
  {
    "message" : "duplicate field selected: Id",
    "errorCode" : "INVALID_FIELD"
  }
]
```

Support for FIELDS()

The `FIELDS ()` keyword is supported in these platform features. For restrictions on what's supported, see [Bounded and Unbounded Queries](#).

- Apex
- The SOQL language wherever query or queryAll operations can be executed.
- Query jobs in Bulk API 2.0.
- The Salesforce CLI.
- The `/query` and `/queryAll` resources in the Lightning Platform REST API.
- The `query()` and `queryAll()` operations in the Lightning Platform SOAP API.

When building SOAP API integrations, take care when adopting `FIELDS ()` with grouping options `STANDARD`, `CUSTOM`, and `ALL`. Any change to an org's object model (such as admin modification, feature enablement, or update via major release) must be immediately reflected in the enterprise WSDL. Otherwise, an error is thrown as the query operation returns a result that doesn't match the contract with the client.

Bounded and Unbounded Queries

The API distinguishes *bounded* queries, which have well-defined sets of fields, from *unbounded* queries, which have sets of fields that the API can't determine in advance. For example, because the number of custom fields for an object isn't predetermined, `FIELDS (CUSTOM)` and `FIELDS (ALL)` are considered unbounded. This table shows the support for `FIELDS ()` in bounded and unbounded queries:

	Bounded – <code>FIELDS(STANDARD)</code>	Unbounded – <code>FIELDS(ALL)</code> and <code>FIELDS(CUSTOM)</code>
Apex (inline and dynamic)	Supported	Not supported
Bulk API 2.0	Supported	Not supported
CLI	Supported	Supported only if the result rows are limited. See Limiting Result Rows .
SOAP API and REST API	Supported	Supported only if the result rows are limited. See Limiting Result Rows .

Limiting Result Rows

To limit the result rows, you can add any of these limits to the query:

- `LIMIT n` where *n* is less than or equal to 200. For example:

```
SELECT FIELDS (ALL) FROM Contact LIMIT 200
```

- `WHERE Id IN idList` where *idList* is list of up to 200 IDs. For example:

```
SELECT FIELDS (ALL) FROM Contact WHERE Id IN ('003R000000ATjnCIAT', '003R000000AZFUIIA5', '003R000000DkYoFIAV')
```

- `WHERE Id IS idList` where *idList* is a list of up to 200 ID tests joined with boolean operators. For example:

```
SELECT FIELDS (ALL) FROM Contact WHERE Id = '003R000000ATjnCIAT' OR Id = '003R000000AZFUIIA5' OR Id = '003R000000DkYoFIAV'
```

Salesforce CLI Example

This example uses `FIELDS ()` with the Salesforce CLI:

```
sf data query --target-org DevHub --query "SELECT FIELDS (STANDARD) FROM Account"
```

This example assumes that the CLI is updated to work with version 51.0 or later of the API.

REST API Example

This request uses `FIELDS ()` with `/query` in REST API:

```
GET https://yourInstance.salesforce.com/services/data/v59.0/query?q=SELECT+FIELDS (STANDARD)+FROM+Account
```


SOAP API Example

This example uses `FIELDS ()` with `query()` in SOAP API:

```
POST https://yourInstance.salesforce.com/services/Soap/c/59.0

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:urn="urn:enterprise.soap.sforce.com">
  <soapenv:Header>
    <urn:SessionHeader>
      <urn:sessionId>sessionId</urn:sessionId>
    </urn:SessionHeader>
  </soapenv:Header>
  <soapenv:Body>
    <urn:query>
      <urn:queryString>SELECT FIELDS(STANDARD) FROM Account</urn:queryString>
    </urn:query>
  </soapenv:Body>
</soapenv:Envelope>
```

Bulk API 2.0 Example

This request uses `FIELDS ()` when creating a bulk query job:

```
POST https://yourInstance.salesforce.com/services/data/v59.0/jobs/query
{
  "operation": "query",
  "query": "SELECT FIELDS(STANDARD) FROM Account"
}
```

Considerations for FIELDS()

Keep these considerations in mind when using the `FIELDS ()` keyword.

- If you already know which fields you want to retrieve, you'll get better performance by specifying them explicitly rather than using `FIELDS ()` and retrieving more fields than you need.
- `FIELDS ()` can cause errors if you use it with operators that require aggregation.
 - For example, without `FIELDS ()` this query works correctly:


```
SELECT Id, MIN(NumberOfEmployees) FROM Account GROUP BY Id
```

 But adding `FIELDS ()` to the query


```
SELECT FIELDS(ALL), MIN(NumberOfEmployees) FROM Account GROUP BY Id LIMIT 200
```

 results in a "Field must be grouped or aggregated" error because it's equivalent to


```
SELECT IsDeleted, <etc.>, MIN(NumberOfEmployees) FROM Account GROUP BY Id LIMIT 200.
```
- If `FIELDS ()` returns a large amount of data, use one of these methods to retrieve all the pages of results:
 - In SOQL, if you use `OFFSET` and `LIMIT` in your query, the number of records returned may be less than `LIMIT`. Check the number of results that were returned and adjust `OFFSET`, as required. Do not just increment `OFFSET` by `LIMIT`.

- In Bulk API 2.0, use the Sforce-Locator response header that is returned in the job's results.
- In REST API, use the nextRecordsUrl that is returned by /query and /queryAll.
- In SOAP API, use queryMore().
- If you use `SELECT FIELDS (CUSTOM)` on an object that doesn't have any custom fields, the query results in an error.

```
HTTP/1.1 400 Bad Request
[
  {
    "message": "'FIELDS(...)' expansion function must result in at least one field
being selected.",
    "errorCode" : "MALFORMED_QUERY"
  }
]
```

If there are any fields included in the field list, there's no error. For example, this query returns status code 200 even if the object contains no custom fields.

```
SELECT Id, FIELDS (CUSTOM) FROM User LIMIT 200
```

- The list of fields returned by `FIELDS ()` reflects the current state of the org's metadata and data model. Any changes to your org's metadata or data model affects query performance and results.

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.

```
[TYPEOF typeOfField
  {WHEN whenObjectType THEN whenFieldList} [...]
  [ELSE elseFieldList]
END] [...]
```

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
<i>typeOfField</i>	A polymorphic relationship field in <i>objectType</i> or a polymorphic field in a parent of <i>objectType</i> that can reference multiple object types. <i>typeOfField</i> cannot reference a relationship field that is also referenced in <i>fieldList</i> .
<i>whenObjectType</i>	An object type for the given <code>WHEN</code> clause. When the <code>SELECT</code> statement runs, each object type associated with the polymorphic relationship field specified in the <i>typeOfField</i> expression is checked for a matching object type in a <code>WHEN</code> clause.

Syntax	Description
<i>whenFieldList</i>	A list of one or more fields, separated by commas, that you want to retrieve from the specified <i>whenObjectType</i> . These are fields in the referenced object type or paths to related object fields, not fields in the primary object type for the <code>SELECT</code> statement.
<i>elseFieldList</i>	A list of one or more fields, separated by commas, that you want to retrieve if none of the <code>WHEN</code> clauses match the object type associated with the polymorphic relationship field specified in <i>typeOfField</i> . This list may only contain fields valid for the Name object type, or paths to related object fields in Name.

Note the following considerations for using `TYPEOF`:

- `TYPEOF` can't be used with a relationship field whose `namePointing` attribute is false.
- `TYPEOF` can't be used with a relationship field whose `relationshipName` 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. For details, see [Filter on Polymorphic Relationship Fields](#).
- `TYPEOF` isn't allowed in queries that don't return objects, such as `COUNT ()`.
- `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 because `TYPEOF` is used in the semi-join query:

```
SELECT Name FROM Account
WHERE CreatedById IN
(
  SELECT
    TYPEOF Owner
      WHEN User THEN Id
      WHEN Group THEN CreatedById
    END
  FROM CASE
)
```

The following example 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
)
```

- TYPEOF can't be used in queries with functions in the `SELECT` clause. The following example is not valid because the `TYPEOF` includes the `FORMAT` function.

```
SELECT
  TYPEOF What
    WHEN Account THEN Id, FORMAT(LastModifiedDate) LastModifiedDate__f
    WHEN Oppty THEN Id
  END
FROM Task
```

Instead, run the same query without functions to retrieve a list of IDs.

```
SELECT
  TYPEOF What
    WHEN Account THEN Id, LastModifiedDate
    WHEN Opportunity THEN Id
  END
FROM Task
```

Then, run a second query with functions on the resulting ID list.

```
SELECT
  FORMAT(LastModifiedDate) LastModifiedDate__f
FROM Account
WHERE Id in RetrievedIdList
```

- TYPEOF can't be used in queries with `GROUP BY`, `GROUP BY ROLLUP`, `GROUP BY CUBE`, and `HAVING`.

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 details on polymorphic relationships, and more examples of `TYPEOF`.

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*.

```
[USING SCOPE filterScope]
```

filterScope can take one of many enumeration values. To get a list of scopes supported by an object, call `describeObject()` for SOAP API or `sObjectDescribe` for REST API. In the `supportedScopes` section of the object's description, the `name` shows the name of the scope and the `label` shows an explanation of that scope. This example shows the description of the Account object.

```
"supportedScopes" : [
  {
    "label" : "All accounts",
    "name" : "everything"
```

```

    },
    {
      "label" : "My accounts",
      "name" : "mine"
    },
    {
      "label" : "My team's accounts",
      "name" : "team"
    }
  ],

```

This table lists examples of enumeration values that `filterScope` can take.

Scope	Description
<code>delegated</code>	Filter for records delegated to another user for action. For example, a query could filter for only delegated Task records.
<code>everything</code>	Filter for all records.
<code>mine</code>	Filter for records owned by the user running the query.
<code>mine_and_my_groups</code>	Filter for records assigned to the user running the query and the user's queues. If a user is assigned to a queue, the user can access records in the queue. This filter applies only to the ProcessInstanceWorkItem object.
<code>my_territory</code>	Filter for records in the territory of the user running the query. This option is available if territory management is enabled for your organization.
<code>my_team_territory</code>	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.
<code>scopingRule</code>	Filter for records based on the applicable scoping rule. This option is available if an admin has activated at least one scoping rule on the object that you're querying.
<code>team</code>	Filter for records assigned to a team, such as an Account team.

WHERE

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

Syntax

```
WHERE conditionExpression
```

conditionExpression

The `conditionExpression` uses the following syntax:

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

 **Note:** Strings in WHERE clauses cannot exceed 4,000 characters.

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 information on date functions, such as CALENDAR_YEAR(), see [Date Functions](#).

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
```

fieldExpression

The field expression syntax of the WHERE clause in a SOQL query consists of a field name, a comparison operator, and a value. The query uses these components to compare the field name value to records being searched.

```
fieldName comparisonOperator value
```

Syntax	Description
<i>fieldName</i>	The name of a field for the specified object. Use of single or double quotes around the name 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. The name doesn't need to be a field in the <i>fieldList</i> .
<i>comparisonOperator</i>	Operators that compare values, such as =, <=, IN, and LIKE. Operators are case insensitive for most fields, but case sensitive for case-sensitive fields.
<i>value</i>	A value used to compare with the value in the <i>fieldName</i> . Supply a value whose data type matches the field type of the specified field. The value must be a valid value, not other field names or calculations. If quotes are required, use single quotes. Double quotes result in an error. Quotes are unnecessary for dates and numbers.

You can use parentheses to define the order in which *fieldExpressions* are evaluated. You must specify parentheses when nesting operators. However, multiple operators of the same type do not need to be nested. In the following example, 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
```

Comparison Operators

SOQL queries can include comparison operators, such as `=`, `<`, `>`, `IN`, and `LIKE` in the field expression of a `WHERE` clause, which you use in a `SELECT` statement. You can also use comparison operators to create complex queries with semi-joins and anti-joins.

The following table lists the ***comparisonOperator*** values that are used in ***fieldExpression*** syntax.

 **Note:** Comparisons on strings are case-sensitive for unique case-sensitive fields and case-insensitive for all other fields.

Operator	Name	Description
<code>=</code>	Equals	Expression is true if the value in the <i>fieldName</i> equals the <i>value</i> in the expression.
<code>!=</code>	Not equals	Expression is true if the value in the <i>fieldName</i> doesn't equal the specified <i>value</i> .
<code><</code>	Less than	Expression is true if the value in the <i>fieldName</i> is less than the specified <i>value</i> .
<code><=</code>	Less or equal	Expression is true if the value in the <i>fieldName</i> is less than or equal to the specified <i>value</i> .
<code>></code>	Greater than	Expression is true if the value in the <i>fieldName</i> is greater than the specified <i>value</i> .
<code>>=</code>	Greater or equal	Expression is true if the value in the <i>fieldName</i> is greater than or equal to the specified <i>value</i> .
<code>LIKE</code>	Like	Expression is true if the value in the <i>fieldName</i> matches the characters of the text string in the specified <i>value</i> . The text string in the specified value must be enclosed in single quotes.

The `LIKE` operator is supported for string fields only. The operator provides a mechanism for matching partial text strings and includes support for:

- The `%` and `_` wildcards.
 - The `%` wildcard matches zero or more characters.
 - The `_` wildcard matches exactly one character.
- The escaping of special characters `%` or `_`.


 **Note:** Don't use the backslash (`\`) character in a search except to escape a special character. See [Quoted String Escape Sequences](#).

This example query matches Appleton, Apple, and Appl, but not Bappl.

```
SELECT AccountId, FirstName, lastname
FROM Contact
WHERE lastname LIKE 'appl%'
```

<code>IN</code>	<code>IN</code>	<p>If the value equals any one of the values in a <code>WHERE</code> clause. The string values for <code>IN</code> must be in parentheses and surrounded by single quotes.</p> <p>You can query values in a field where another field on the same object has a specified set of values, using <code>IN</code>. For example:</p>
-----------------	-----------------	---

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

Operator	Name	Description
		You can also use <code>IN</code> and <code>NOT IN</code> for semi-joins and anti-joins when querying on ID (primary key) or reference (foreign key) fields.
<code>NOT IN</code>	<code>NOT IN</code>	<p>If the value doesn't equal any of the values in a <code>WHERE</code> clause. The string values for <code>NOT IN</code> must be in parentheses and surrounded by single quotes. For example:</p> <pre>SELECT Name FROM Account WHERE BillingState NOT IN ('California', 'New York')</pre> <p> Note: The logical operator <code>NOT</code> is unrelated to this comparison operator.</p>
<code>INCLUDES</code> <code>EXCLUDES</code>		Applies only to multi-select picklists. See Query Multi-Select Picklists .

Semi-Joins with `IN` and Anti-Joins with `NOT IN`

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. 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.

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 don't 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 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's compared. For restrictions that prevent unnecessary processing of semi-join queries, see [Considerations](#).

Reference Field Semi-Join

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

```
SELECT Id
FROM Task
```



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

Notice that the left operand, `WhoId`, of the `IN` clause is a reference field. An interesting aspect of this query is that `WhoId` 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 don't 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 isn't `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
)
```



Note: 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
)
```

Considerations for Semi-Join and Anti-Join Queries

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:**

- You can't use more than two `IN` or `NOT IN` statements per `WHERE` clause.
- You can't 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's 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 can't traverse relationships. This limit means that you can't 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` isn't supported. Similarly, subqueries like `Contact.AccountId` `FROM Case` are invalid.

- You can't 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'
)
```

Rewrite the query in a valid form, for example:

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

- You can't nest a semi-join or anti-join statement in another semi-join or anti-join statement.
- You can't use semi-joins and anti-joins in a `HAVING` clause.
- 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:

```
SELECT Id
FROM Idea
WHERE (Idea.Title LIKE 'Vacation%')
AND (Idea.LastCommentDate > YESTERDAY)
AND (Id IN (SELECT ParentId FROM Vote
            WHERE CreatedById = '005x0000000sMgYAAU'
            AND Parent.Type='Idea'))
```

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

```
SELECT Id
FROM Idea
WHERE
    ((Idea.Title LIKE 'Vacation%')
    AND (CreatedDate > YESTERDAY)
    AND (Id IN (SELECT ParentId FROM Vote
                WHERE CreatedById = '005x0000000sMgYAAU'
                AND Parent.Type='Idea'))
```

```
)
OR (Idea.Title like 'ExcellentIdea%'))
```

- You can't use subqueries with OR.
- COUNT, FOR UPDATE, ORDER BY, and LIMIT aren't supported in subqueries.
- The following objects aren't currently supported in subqueries:
 - ActivityHistory
 - Attachments
 - Event
 - 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	<i>fieldExpressionX</i> AND <i>fieldExpressionY</i>	true if both <i>fieldExpressionX</i> and <i>fieldExpressionY</i> are true.
OR	<i>fieldExpressionX</i> OR <i>fieldExpressionY</i>	<p>true if either <i>fieldExpressionX</i> or <i>fieldExpressionY</i> is true.</p> <p>In a WHERE clause that uses OR, records are returned even if the foreign key value in a record is null.</p> <pre>SELECT Id FROM Contact WHERE LastName = 'Young' or Account.Name = 'Quarry'</pre>
NOT	not <i>fieldExpressionX</i>	<p>true if <i>fieldExpressionX</i> is false.</p> <p>There is also a comparison operator NOT IN, which is different from this logical operator.</p>

Date Formats and Date Literals in WHERE

You can specify date values or date literals in WHERE clauses to filter SOQL query results. Dates represent a specific day or time, while date literals represent a relative range of time, such as last month, this week, or next year.

For information on formatting the dates and times returned by Salesforce, see [FORMAT \(\)](#) and [convertTimezone\(\)](#).

Filter Query Results Using Dates

The *fieldExpression* of a `WHERE` clause supports filtering query results based on `date` and `dateTime` fields. Use comparison operators between a field name and a specified date value to filter for results that match the condition. For example, this query filters for Account records that were created after the specified date and time.

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

To filter query results based on a specific date, set a value using a `date` format. To filter query results based on a specific date and time, set a value using a `dateTime` format.

This table shows the supported `date` and `dateTime` formats that you can use in the `WHERE` clause of an SOQL query.

Field Type	Format	Example
<code>date</code>	YYYY-MM-DD	1999-01-01
<code>dateTime</code>	<ul style="list-style-type: none"> YYYY-MM-DDThh:mm:ss+hh:mm YYYY-MM-DDThh:mm:ss-hh:mm YYYY-MM-DDThh:mm:ssZ 	<ul style="list-style-type: none"> 1999-01-01T23:01:01+01:00 1999-01-01T23:01:01-08:00 1999-01-01T23:01:01Z

Although `dateTime` field values are stored in Salesforce as Coordinated Universal Time (UTC), you can use the time zone offset to provide `dateTime` values in another timezone. The time zone offset is always from UTC.

For more information on `dateTime` formats and time zone offsets, see:

- <http://www.w3.org/TR/xmlschema-2/#isoformats>
- <http://www.w3.org/TR/NOTE-datetime>

Filter Query Results Using Date Literals

You can use a date literal in the *fieldExpression* of a `WHERE` clause to filter query results relative to a range of dates. For example, you can filter for query results that were created within the last three months, or you can filter for results with deadlines after the next fiscal year.


Each date literal represents a range of time relative to the current day. The exact start and stop of the range depends on the date literal and the locale of the user sending the query. If the user hasn't set a personal locale, the range depends on the locale of the org. For more information, see [Select Your Language, Locale, and Currency in Salesforce Help](#).

To filter for results within a range, use the `=` comparison operator. To filter for results on either side of a range, use the `>` or `<` comparison operators. The table shows the available list of date literals, the ranges they represent, and examples.

Date Literal	Range	Example
YESTERDAY	Starts at 12:00:00 AM on the day before the current day and continues for 24 hours.	<code>SELECT Id FROM Account WHERE CreatedDate = YESTERDAY</code>
TODAY	Starts at 12:00:00 AM on the current day and continues for 24 hours.	<code>SELECT Id FROM Account WHERE CreatedDate > TODAY</code>

Date Literal	Range	Example
TOMORROW	Starts at 12:00:00 AM on the day after the current day and continues for 24 hours.	SELECT Id FROM Opportunity WHERE CloseDate = TOMORROW
LAST_WEEK	Starts at 12:00:00 AM on the first day of the week before the current week and continues for seven days.	SELECT Id FROM Account WHERE CreatedDate > LAST_WEEK
THIS_WEEK	Starts at 12:00:00 AM on the first day of the current week and continues for seven days.	SELECT Id FROM Account WHERE CreatedDate < THIS_WEEK
NEXT_WEEK	Starts at 12:00:00 AM on the first day of the week after the current week and continues for seven days.	SELECT Id FROM Opportunity WHERE CloseDate = NEXT_WEEK
LAST_MONTH	Starts at 12:00:00 AM on the first day of the month before the current month and continues for all the days of that month.	SELECT Id FROM Opportunity WHERE CloseDate > LAST_MONTH
THIS_MONTH	Starts at 12:00:00 AM on the first day of the current month and continues for all the days of that month.	SELECT Id FROM Account WHERE CreatedDate < THIS_MONTH
NEXT_MONTH	Starts at 12:00:00 AM on the first day of the month after the current month and continues for all the days of that month.	SELECT Id FROM Opportunity WHERE CloseDate = NEXT_MONTH
LAST_90_DAYS	Starts at 12:00:00 AM 90 days before the current day and continues up to the current second. (The range includes today. Using this date value includes records from 91 days ago up to the current day.)	SELECT Id FROM Account WHERE CreatedDate = LAST_90_DAYS
NEXT_90_DAYS	Starts at 12:00:00 AM on the day that the report is run and continues for 90 days. (The range includes today.)	SELECT Id FROM Opportunity WHERE CloseDate > NEXT_90_DAYS
LAST_N_DAYS:n	Starts at 12:00:00 AM <i>n</i> days before the current day and continues up to the current second. (The range includes today. Using this date value includes records from <i>n</i> + 1 days ago up to the current day.) In standard filters, <i>n</i> can be 7, 30, 60, 90, or 120.	SELECT Id FROM Account WHERE CreatedDate = LAST_N_DAYS:365
NEXT_N_DAYS:n	For standard date filters, starts at 12:00:00 AM on the day that the report is run and continues for <i>n</i> days. (The range includes today.) In standard filters, <i>n</i> can be 7, 30, 60, 90, or 120. For custom field filters, starts at 12:00:00 AM on the next day and continues for the next <i>n</i> days. (The range doesn't include today.)	SELECT Id FROM Opportunity WHERE CloseDate > NEXT_N_DAYS:15

Date Literal	Range	Example
<code>N_DAYS_AGO:n</code>	Starts at 12:00:00 AM on the day <i>n</i> days before the current day and continues for 24 hours. (The range doesn't include today.)	<code>SELECT Id FROM Opportunity WHERE CloseDate = N_DAYS_AGO:25</code>
<code>NEXT_N_WEEKS:n</code>	Starts at 12:00:00 AM on the first day of the week after the current week and continues for <i>n</i> times seven days.	<code>SELECT Id FROM Opportunity WHERE CloseDate > NEXT_N_WEEKS:4</code>
<code>LAST_N_WEEKS:n</code>	Starts at 12:00:00 AM on the first day of the week that started <i>n</i> weeks before the current week, and continues up to 11:59 PM on the last day of the week before the current week.	<code>SELECT Id FROM Account WHERE CreatedDate = LAST_N_WEEKS:52</code>
<code>N_WEEKS_AGO:n</code>	Starts at 12:00:00 AM on the first day of the week that started <i>n</i> weeks before the start of the current week and continues for seven days.	<code>SELECT Id FROM Opportunity WHERE CloseDate = N_WEEKS_AGO:3</code>
<code>NEXT_N_MONTHS:n</code>	Starts at 12:00:00 AM on the first day of the month after the current month and continues until the end of the <i>n</i> th month.	<code>SELECT Id FROM Opportunity WHERE CloseDate > NEXT_N_MONTHS:2</code>
<code>LAST_N_MONTHS:n</code>	Starts at 12:00:00 AM on the first day of the month that started <i>n</i> months before the current month and continues up to 11:59 PM on the last day of the month before the current month.	<code>SELECT Id FROM Account WHERE CreatedDate = LAST_N_MONTHS:12</code>
<code>N_MONTHS_AGO:n</code>	Starts at 12:00:00 AM on the first day of the month that started <i>n</i> months before the start of the current month and continues for all the days of that month.	<code>SELECT Id FROM Opportunity WHERE CloseDate = N_MONTHS_AGO:6</code>
<code>THIS_QUARTER</code>	Starts at 12:00:00 AM on the first day of the current calendar quarter and continues to the end of the quarter.	<code>SELECT Id FROM Account WHERE CreatedDate = THIS_QUARTER</code>
<code>LAST_QUARTER</code>	Starts at 12:00:00 AM on the first day of the calendar quarter before the current calendar quarter and continues to the end of that quarter.	<code>SELECT Id FROM Account WHERE CreatedDate = LAST_QUARTER</code>
<code>NEXT_QUARTER</code>	Starts at 12:00:00 AM on the first day of the calendar quarter after the current calendar quarter and continues to the end of that quarter.	<code>SELECT Id FROM Account WHERE CreatedDate < NEXT_QUARTER</code>
<code>NEXT_N_QUARTERS:n</code>	Starts at 12:00:00 AM on the first day of the calendar quarter after the current quarter and continues to the end of the calendar quarter <i>n</i> quarters in the future. (The range doesn't include the current quarter.)	<code>SELECT Id FROM Account WHERE CreatedDate < NEXT_N_QUARTERS:2</code>
<code>LAST_N_QUARTERS:n</code>	Starts at 12:00:00 AM on the first day of the calendar quarter <i>n</i> quarters ago and continues to the end of the calendar quarter before the current	<code>SELECT Id FROM Account WHERE CreatedDate = LAST_N_QUARTERS:2</code>

Date Literal	Range	Example
	quarter. (The range doesn't include the current quarter.)	
<code>N_QUARTERS_AGO:n</code>	Starts at 12:00:00 AM on the first day of the calendar quarter <i>n</i> quarters before the current calendar quarter and continues to the end of that quarter.	<code>SELECT Id FROM Opportunity WHERE CloseDate = N_QUARTERS_AGO:3</code>
<code>THIS_YEAR</code>	Starts at 12:00:00 AM on January 1 of the current year and continues through the end of December 31 of the current year.	<code>SELECT Id FROM Opportunity WHERE CloseDate = THIS_YEAR</code>
<code>LAST_YEAR</code>	Starts at 12:00:00 AM on January 1 of the year before the current year and continues through the end of December 31 of that year.	<code>SELECT Id FROM Opportunity WHERE CloseDate > LAST_YEAR</code>
<code>NEXT_YEAR</code>	Starts at 12:00:00 AM on January 1 of the year after the current year and continues through the end of December 31 of that year.	<code>SELECT Id FROM Opportunity WHERE CloseDate < NEXT_YEAR</code>
<code>NEXT_N_YEARS:n</code>	Starts at 12:00:00 AM on January 1 of the year after the current year and continues through the end of December 31 of the <i>n</i> th year.	<code>SELECT Id FROM Opportunity WHERE CloseDate = NEXT_N_YEARS:5</code>
<code>LAST_N_YEARS:n</code>	Starts at 12:00:00 am on January 1, <i>n+1</i> years ago. The range ends on December 31 of the year before the current year.	<code>SELECT Id FROM Opportunity WHERE CloseDate = LAST_N_YEARS:5</code>
<code>N_YEARS_AGO:n</code>	Starts at 12:00:00 AM on January 1 of the calendar year <i>n</i> years before the current calendar year and continues through the end of December 31 of that year.	<code>SELECT Id FROM Opportunity WHERE CloseDate = N_YEARS_AGO:2</code>
<code>THIS_FISCAL_QUARTER</code>	Starts at 12:00:00 AM on the first day of the current fiscal quarter and continues through the end of the last day of the current fiscal quarter. The fiscal quarter is defined on the Fiscal Year page in Setup.	<code>SELECT Id FROM Account WHERE CreatedDate = THIS_FISCAL_QUARTER</code>
<code>LAST_FISCAL_QUARTER</code>	Starts at 12:00:00 AM on the first day of the fiscal quarter before the current fiscal quarter and continues through the last day of that fiscal quarter. The fiscal quarter is defined on the Fiscal Year page in Setup.	<code>SELECT Id FROM Account WHERE CreatedDate > LAST_FISCAL_QUARTER</code>
	 Note: None of the FISCAL literal date values are supported when creating mobile custom views.	
<code>NEXT_FISCAL_QUARTER</code>	Starts at 12:00:00 AM on the first day of the fiscal quarter after the current fiscal quarter and continues through the last day of that fiscal quarter. (The range doesn't include the current	<code>SELECT Id FROM Account WHERE CreatedDate < NEXT_FISCAL_QUARTER</code>

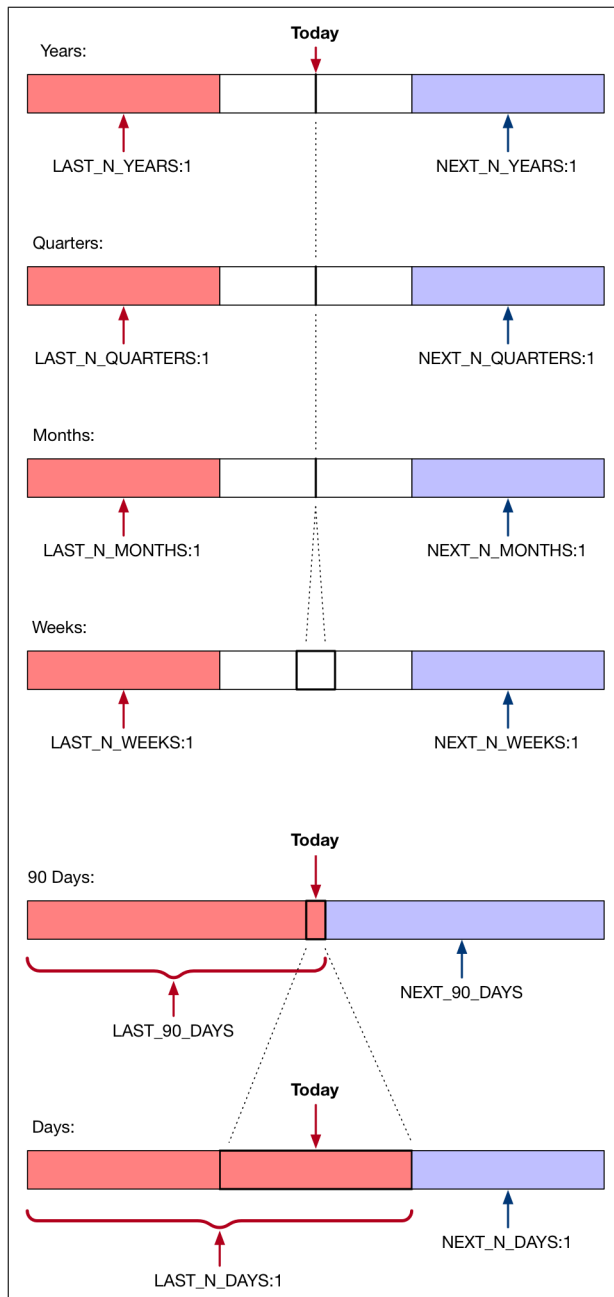
Date Literal	Range	Example
	quarter.) The fiscal quarter is defined on the Fiscal Year page in Setup.	
<code>NEXT_N_FISCAL_</code> <code>QUARTERS : n</code>	Starts at 12:00:00 AM on the first day of the fiscal quarter after the current fiscal quarter and continues through the end of the last day of the <i>n</i> th fiscal quarter. (The range doesn't include the current fiscal quarter.) The fiscal quarter is defined on the Fiscal Year page in Setup.	<code>SELECT Id FROM Account WHERE CreatedDate = NEXT_N_FISCAL_QUARTERS : 6</code>
<code>LAST_N_FISCAL_</code> <code>QUARTERS : n</code>	Starts at 12:00:00 AM on the first day of the fiscal quarter <i>n</i> fiscal quarters ago and continues through the end of the last day of the previous fiscal quarter. (The range doesn't include the current fiscal quarter.) The fiscal quarter is defined on the Fiscal Year page in Setup.	<code>SELECT Id FROM Account WHERE CreatedDate > LAST_N_FISCAL_QUARTERS : 6</code>
<code>N_FISCAL_QUARTERS_AGO : n</code>	Starts at 12:00:00 AM on the first day of the fiscal quarter <i>n</i> fiscal quarters before the current fiscal quarter and continues through the end of the last day of that fiscal quarter.	<code>SELECT Id FROM Opportunity WHERE CloseDate = N_FISCAL_QUARTERS_AGO : 6</code>
<code>THIS_FISCAL_YEAR</code>	Starts at 12:00:00 AM on the first day of the current fiscal year and continues through the end of the last day of the fiscal year. The fiscal quarter is defined on the Fiscal Year page in Setup.	<code>SELECT Id FROM Opportunity WHERE CloseDate = THIS_FISCAL_YEAR</code>
<code>LAST_FISCAL_YEAR</code>	Starts at 12:00:00 AM on the first day of the fiscal year before the current fiscal year and continues through the end of the last day of that fiscal year. The fiscal quarter is defined on the Fiscal Year page in Setup.	<code>SELECT Id FROM Opportunity WHERE CloseDate > LAST_FISCAL_YEAR</code>
<code>NEXT_FISCAL_YEAR</code>	Starts at 12:00:00 AM on the first day of the fiscal year after the current fiscal year and continues through the end of the last day of that fiscal year. The fiscal quarter is defined on the Fiscal Year page in Setup.	<code>SELECT Id FROM Opportunity WHERE CloseDate < NEXT_FISCAL_YEAR</code>
<code>NEXT_N_FISCAL_</code> <code>YEARS : n</code>	Starts at 12:00:00 AM on the first day of the fiscal year after the current fiscal year and continues through the end of the last day of the <i>n</i> th fiscal year. (The range doesn't include the current fiscal year.) The fiscal quarter is defined on the Fiscal Year page in Setup.	<code>SELECT Id FROM Opportunity WHERE CloseDate = NEXT_N_FISCAL_YEARS : 3</code>
<code>LAST_N_FISCAL_</code> <code>YEARS : n</code>	Starts at 12:00:00 AM on the first day of the fiscal year <i>n</i> fiscal years ago and continues through the end of the last day of the fiscal year before the current fiscal year. (The range doesn't include the	<code>SELECT Id FROM Opportunity WHERE CloseDate = LAST_N_FISCAL_YEARS : 3</code>

Date Literal	Range	Example
	current fiscal year.) The fiscal quarter is defined on the Fiscal Year page in Setup.	
<code>N_FISCAL_YEARS_AGO:n</code>	Starts at 12:00:00 AM on the first day of the fiscal year <i>n</i> fiscal years ago and continues through the end of the last day of that fiscal year.	<code>SELECT Id FROM Opportunity WHERE CloseDate = N_FISCAL_YEARS_AGO:3</code>

To use `FISCAL` date literals, you must first define custom fiscal years in Salesforce. If you use `FISCAL` date literals and specify a time range outside of a defined fiscal year, an invalid date error is returned.

Whether the current day is included in the range of a date literal depends on which date literal you use.

- If the `UNIT` in `LAST_N_UNIT:n` is any unit except `DAYS`, the date literal doesn't include the current day. So, for example, `LAST_N_WEEKS:1` doesn't include today.
- The two date literals `LAST_N_DAYS:n` and `LAST_90_DAYS` *do* include the current day. So, for example, `LAST_N_DAYS:1` includes yesterday and today. And `LAST_90_DAYS` includes 91 days, not just 90.



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.

Filter on Polymorphic Relationship Fields

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

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 wouldn't 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 details on polymorphic relationships, and more filtering examples.

Query 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, which allows selection of multiple items. In API version 39.0 and later, query for picklist values by the value's API name, which can be different from the actual value.

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

Operator	Description
<code>=</code>	Equals the specified string.
<code>!=</code>	Does not equal the specified string.
<code>includes</code>	Contains the specified string.
<code>excludes</code>	Does not contain the specified string.
<code>;</code>	Specifies AND for two or more strings. Use <code>;</code> for multi-select picklists when two or more items must be selected. For example: <div>'AAA;BBB'</div>

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 '
```

Use `null` in WHERE

You can search for null values by using the `null` keyword in SOQL queries.

This example query returns the account IDs of all events with a non-null activity date.

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

If you run a query on a boolean field, null matches FALSE values. For example, if `Test_c` is a boolean field, then the following query returns all the account records where `Test_c` is false:

```
SELECT Id, Name Test_c
FROM Account
WHERE Test_c = null
```

The clause `WHERE Test_c = null` is equivalent to `WHERE Test_c = false`. The clause `WHERE Test_c != null` is equivalent to `WHERE Test_c = true`.

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, the record is returned even if the parent does not exist.

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

Case record `Id` values are returned.

SEE ALSO:

[null Values in Lookup Relationships and Outer Joins](#)

WITH

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 **WITH *filteringExpression***. 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](#).
- 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](#) in the *Apex Developer Guide*.
- To specify user mode access for Apex database operations. Apex code runs in system mode by default, which means that it runs with substantially elevated permissions over the user running the code. To enhance the security context of Apex, you can specify user mode access by using `WITH USER_MODE`. Field-level security (FLS) and object permissions of the running user are respected in user mode, unlike in system mode. User mode always applies sharing rules, but in system mode they're controlled by sharing keywords on the class. See [Enforce User Mode for Database Operations \(Beta\)](#) 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.


- `SELECT Title FROM KnowledgeArticleVersion WHERE PublishStatus='online' WITH DATA CATEGORY Geography__c ABOVE usa__c`
- `SELECT Id FROM UserProfileFeed WITH UserId='005D00000001AamR' ORDER BY CreatedDate DESC, Id DESC LIMIT 20`

WITH DATA CATEGORY

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.

```
[WITH [DATA CATEGORY] filteringExpression]
```

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.

 **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” in Salesforce Help.

 **Note:** The WITH DATA CATEGORY clause does not support [bind variables](#). The clause can have no more than three data category conditions.

filteringExpression

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

```
dataCategorySelection [AND [dataCategorySelection] [...]]
```

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.

- 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:

```
dataCategoryGroupName filteringSelector dataCategoryName
```

Syntax	Description
<i>dataCategoryGroupName</i>	<p>The name of the data category group to use as a filter. <code>Geography__c</code> is the data category group in the following example.</p> <p>You cannot use the same data category group more than once in a query. As an example, the following command is incorrect: <code>WITH DATA CATEGORY Geography__c ABOVE usa__c AND Geography__c BELOW europe__c</code></p>
<i>filteringSelector</i>	<p>The selector used to filter the data in the specified data category. See Filtering Selectors for a list of valid selectors.</p>
<i>dataCategoryName</i>	<p>The name of the data category for filtering. You must have visibility on the category you specify.</p> <p>You can use parentheses to apply the filtering operator to more than one data category. Each data category must be separated by a comma.</p> <p>Example: <code>WITH DATA CATEGORY Geography__c AT (usa__c,france__c,uk__c)</code></p> <p>You can't use the <code>AND</code> operator instead of parentheses to list multiple data categories. The following syntax does not work <code>WITH DATA CATEGORY Geography__c AT usa__c AND france__c</code></p>

Filtering Selectors

When specifying filters for a `WITH DATA 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 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

```

This table lists the *filteringSelector* values that are used in the [dataCategorySelection](#) syntax.

Selector	Description
<code>AT</code>	<p>Select the specified data category.</p> <p>For example, the following syntax selects <code>asia__c</code>.</p> <pre>WITH DATA CATEGORY Geography__c AT asia__c</pre>

Selector	Description
ABOVE	<p>Select the specified data category and all its parent categories.</p> <p>For example, the following syntax selects <code>usa__c</code>, <code>northAmerica__c</code>, and <code>ww__c</code>.</p> <pre>WITH DATA CATEGORY Geography__c ABOVE usa__c</pre>
BELOW	<p>Select the specified data category and all its subcategories.</p> <p>For example, the following selects <code>northAmerica__c</code>, <code>usa__c</code>, <code>canada__c</code>, and <code>mexico__c</code>.</p> <pre>WITH DATA CATEGORY Geography__c BELOW northAmerica__c</pre>
ABOVE_OR_BELOW	<p>Select the specified data category and</p> <ul style="list-style-type: none"> all its parent categories all its subcategories <p>For example, the following selects <code>ww__c</code>, <code>europe__c</code>, <code>france__c</code>, and <code>uk__c</code>.</p> <pre>WITH DATA CATEGORY Geography__c ABOVE_OR_BELOW europe__c</pre>

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

Type of Search	Examples
Select the title from all questions classified with the <code>mobile_phones__c</code> data category in the <code>Product__c</code> data category group	<pre>SELECT Title FROM Question WHERE LastReplyDate < 2005-10-08T01:02:03Z WITH DATA CATEGORY Product__c AT mobile_phones__c</pre>
<p>Select the title and summary from all published Knowledge articles classified:</p> <ul style="list-style-type: none"> above or below <code>europe__c</code> in the <code>Geography__c</code> data category group below <code>allProducts__c</code> in the <code>Product__c</code> data category group 	<pre>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</pre>
<p>Select the ID and title from draft articles of type "Offer__kav" classified :</p> <ul style="list-style-type: none"> with the <code>france__c</code> or <code>usa__c</code> data category in the <code>Geography__c</code> data category group above the <code>dsl__c</code> data category in the <code>Product__c</code> data category group 	<pre>SELECT Id, Title FROM Offer__kav WHERE PublishStatus='Draft' AND Language = 'en_US' WITH DATA CATEGORY Geography__c AT (france__c,usa__c) AND Product__c ABOVE dsl__c</pre>

WITH RecordVisibilityContext

You can use `RecordVisibilityContext` to filter `WITH` clauses to query the attributes that determine the visibility of one or more records. This feature is available in API version 48.0 and later.

The following fields can be used as parameters:

Field	Description	Default
<code>maxDescriptorPerRecord</code>	The maximum number of descriptors to return per record. If the actual number of descriptors for a specific record exceeds that value, then a "tooManyDescriptors" visibility attribute is returned for that record.	<ul style="list-style-type: none"> orgs with CRM Analytics: 150 other orgs: 400
<code>supportsDomains</code>	Whether the Visibility Service is allowed to generate "domain" visibility attributes.	true
<code>supportsDelegates</code>	Whether the Visibility Service is allowed to generate "delegate" visibility attributes.	true

For example:

```
SELECT Id, RecordVisibility.VisibilityAttribute FROM Account WHERE Id = 'xxx'
  WITH RecordVisibilityContext (maxDescriptorPerRecord=100, supportsDomains=true,
supportsDelegates=true)
```

```
SELECT recordId, VisibilityAttribute FROM RecordVisibility WHERE recordId = 'xxx'
  WITH RecordVisibilityContext (maxDescriptorPerRecord=100, supportsDomains=true,
supportsDelegates=true)
```

At least one field is necessary. This query is *invalid*:

```
SELECT Id, RecordVisibility.VisibilityAttribute FROM Account WHERE Id = 'xxx'
  WITH RecordVisibilityContext
```

This query is valid:

```
SELECT Id, RecordVisibility.VisibilityAttribute FROM Account WHERE Id = 'xxx'
  WITH RecordVisibilityContext (maxDescriptorPerRecord=100)
```

For more information, see [RecordVisibility](#) in the *Object Reference*.

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.

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.

```
[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
```

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
```

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.

You can use `GROUP BY ROLLUP` with [aggregate functions](#), such as `SUM()` and `COUNT(fieldName)`.

```
[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.

Note:

- 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)
```

This table shows the query results that include first-level subtotals and grand total rows. 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.

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	Web	4	Four leads with Status = Working - Contacted and LeadSource = Web
Working - Contacted	Phone Inquiry	1	One lead with Status = Working - Contacted and LeadSource = Phone Inquiry
Working - Contacted	Partner Referral	3	Three leads with Status = Working - Contacted and LeadSource = Partner Referral
Working - Contacted	Purchased List	4	Four leads with Status = Working - Contacted and LeadSource = Purchased List
Working - Contacted	null	12	First-level subtotal for all leads with Status = Working - Contacted

Status	LeadSource	cnt	Comment
Closed - Converted	Web	1	One lead with Status = Closed - Converted and LeadSource = Web
Closed - Converted	Phone Inquiry	1	One lead with Status = Closed - Converted and LeadSource = Phone Inquiry
Closed - Converted	Purchased List	1	One lead with Status = Closed - Converted and LeadSource = Purchased List
Closed - Converted	null	3	First-level subtotal for all leads with Status = Closed - Converted
Closed - Not Converted	Web	1	One lead with Status = Closed - Not Converted and LeadSource = Web
Closed - Not Converted	Phone Inquiry	1	One lead with Status = Closed - Not Converted and LeadSource = Phone Inquiry
Closed - Not Converted	Partner Referral	1	One lead with Status = Closed - Not Converted and LeadSource = Partner Referral
Closed - Not Converted	Purchased List	1	One lead with Status = Closed - Not Converted and LeadSource = Purchased List
Closed - Not Converted	null	4	First-level subtotal for all leads with Status = Closed - Not Converted
null	null	22	Grand total of 22 leads

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.

You can use `GROUP BY CUBE` with [aggregate functions](#), such as `SUM()` and `COUNT(fieldName)`.

```
[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, 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.

Type	BillingCountry	grpType	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
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

Use 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 `expri`, where *i* denotes the order of the aggregated fields with no explicit aliases. The value of *i* starts at 0 and increments for every aggregated field with no explicit alias.

In this 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 this 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
```

Identify Subtotals with GROUP BY

When you use the `GROUP BY ROLLUP` or `GROUP BY CUBE` clause in a SOQL query to add the subtotals, you can identify whether the row is a subtotal for a field using the `GROUPING (fieldName)` function.

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.

This example 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.

```
SELECT LeadSource, Rating,
       GROUPING(LeadSource) grpLS, GROUPING(Rating) grpRating,
       COUNT(Name) cnt
FROM Lead
GROUP BY ROLLUP(LeadSource, Rating)
```

The following table shows the query results.

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)
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)`.

Considerations for GROUP BY

Understand the special behavior and limitations for using the `GROUP BY` clause in SOQL queries.

- Some object fields have a field type that doesn't support grouping. You can't include fields with these field types in a `GROUP BY` clause.

To check whether a field supports grouping, you can use SOAP API or REST API to make a `describe` call for the object that contains the field. For each field, the response includes a boolean `groupable` attribute that defines whether you can include the field in a `GROUP BY` clause. To check whether a field supports grouping for standard objects, you can also check the properties for the field in the *Object Reference for Salesforce and Lightning Platform*.

- You must use a `GROUP BY` clause if your query has 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's no `GROUP BY` clause:

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

- You can't use child relationship expressions that use the `__r` syntax in a query that uses a `GROUP BY` clause. For more information, see [Understanding Relationship Names, Custom Objects, and Custom Fields](#).
- In SOAP API, queries that include a `GROUP BY` clause can't use the `queryMore()` call to get more results.
- In REST API, queries that include a `GROUP BY` clause can't use the [query locator](#) to get more results.

HAVING

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

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.

Similar to a `WHERE` clause, the `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.

```
[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
```

This example query returns accounts with duplicate names.

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

This example query 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%'
```

 **Note:** A `HAVING` clause can't contain any semi-join or anti-join subqueries.

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.

```
[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 or NULLS LAST	Orders null records at the beginning (NULLS FIRST) or end (NULLS LAST) of the results. By default, null values are sorted first.

There's no guarantee of the order of results unless you use an `ORDER BY` clause in a query.

And even if you use `ORDER BY`, the order of results can vary if there are duplicate values for the fields you're using in the `ORDER BY` clause. For example, if there are multiple Account records with the same Industry, the order of the results for this query can vary:

```
SELECT Name, Industry FROM Account ORDER BY Industry
```

To avoid this issue, add the `Id` (or any other field that is unique in the results) to the `ORDER BY` clause. For example:

```
SELECT Name, Industry FROM Account ORDER BY Industry, Id
```

The following example 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`:

- `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, the record is returned even if the foreign key value in a record is null.

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

Any case record for which `AccountId` is empty is returned.

- The sort order depends on your user locale setting. For English locales, SOQL uses the UTF-8 values of the uppercase character to create a sort order. In other words, sorting for English locales is case insensitive.

For non-English locales, SOQL uses a pre-defined order that is natural for the locale specified. This is because different cultures may sort the same characters in different ways, and this nuance cannot be captured by the UTF-8 values alone.

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.

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 a starting row in the result set returned by a 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. The result set for the example query would be a subset of the full result set, returning rows 11 through 110 of the full set.

```
SELECT Name
FROM Merchandise__c
WHERE Price__c > 5.0
ORDER BY Name
LIMIT 100
OFFSET 10
```

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.
- The `OFFSET` clause is allowed in SOQL used in SOAP API, REST API, and Apex. It's not allowed in SOQL used within Bulk APIs or Streaming API.
- `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 Opportunities 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.
- 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 and query locator are created to cache and find 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 and consistently locate specific records in the database, use the [queryMore\(\)](#) call.

- 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 is 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.

FOR VIEW and FOR REFERENCE

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. To update recent usage data for retrieved objects, use the `FOR VIEW` clause in conjunction with the `FOR REFERENCE` clause.

FOR VIEW

Use the optional `FOR VIEW` clause in a SOQL query to update objects with information about when they were last viewed.

When the `FOR VIEW` 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.

This example SOQL query retrieves a contact 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

Use the optional `FOR REFERENCE` 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.

When the `FOR REFERENCE` 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.

This example SOQL query 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
```



Note:

- Use the `FOR VIEW` and `FOR REFERENCE` clauses only when you are sure that the retrieved records will definitely be viewed by the logged-in user, else the clause incorrectly updates the usage information for the records. Also, the user won't recognize any incorrectly 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.

UPDATE

Track keywords that are used in Salesforce Knowledge article searches by using the `UPDATE TRACKING` optional clause. Determine how many hits a Salesforce Knowledge article has had by using the `UPDATE VIEWSTAT` optional clause.

Update an Article's Keyword Tracking with SOQL

`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.

```
SELECT Title FROM FAQ__kav
WHERE Keyword='Apex' and
Language = 'en_US' and
KnowledgeArticleVersion = 'ka230000000PCiy'
UPDATE TRACKING
```

Update an Article Viewstat with SOQL

The `UPDATE VIEWSTAT` clause is used in a `SELECT` statement to report on Salesforce Knowledge article searches and views. You can get a view count for every article that you have access to online. Developers can use `UPDATE VIEWSTAT` 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 Knowledge__kav
WHERE PublishStatus='online' and
Language = 'en_US' and
KnowledgeArticleVersion = 'ka230000000PCiy'
UPDATE VIEWSTAT
```

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.



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

To lock a set of sObject records in Apex, embed the keywords `FOR UPDATE` after any inline SOQL statement. This example queries for two accounts and also locks the accounts that are returned.

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





Warning: Use care when setting locks in your Apex code. For details, see [Locking Records](#) in the *Apex Developer Guide*.

SOQL SELECT Examples

The following are examples of text searches that use SOQL.

Type of Search	Examples
Simple query	<code>SELECT Id, Name, BillingCity FROM Account</code>
WHERE	<code>SELECT Id FROM Contact WHERE Name LIKE 'A%' AND MailingCity = 'California'</code>
ORDER BY	<code>SELECT Name FROM Account ORDER BY Name DESC NULLS LAST</code>
LIMIT	<code>SELECT Name FROM Account WHERE Industry = 'media' LIMIT 125</code>
ORDER BY with LIMIT	<code>SELECT Name FROM Account WHERE Industry = 'media' ORDER BY BillingPostalCode ASC NULLS LAST LIMIT 125</code>
count()	<code>SELECT COUNT() FROM Contact</code>
GROUP BY	<code>SELECT LeadSource, COUNT(Name) FROM Lead GROUP BY LeadSource</code>
HAVING	<code>SELECT Name, COUNT(Id) FROM Account GROUP BY Name HAVING COUNT(Id) > 1</code>
OFFSET with ORDER BY	<code>SELECT Name, Id FROM Merchandise__c ORDER BY Name OFFSET 100</code>
OFFSET with ORDER BY and LIMIT	<code>SELECT Name, Id FROM Merchandise__c ORDER BY Name LIMIT 20 OFFSET 100</code>
Relationship queries: child-to-parent	<code>SELECT Contact.FirstName, Contact.Account.Name FROM Contact</code> <code>SELECT Id, Name, Account.Name FROM Contact WHERE Account.Industry = 'media'</code>
Relationship queries: parent-to-child	<code>SELECT Name, (SELECT LastName FROM Contacts) FROM Account</code> <code>SELECT Account.Name, (SELECT Contact.LastName FROM Account.Contacts) FROM Account</code>
Relationship query with WHERE	<code>SELECT Name, (SELECT LastName FROM Contacts WHERE CreatedBy.Alias = 'x') FROM Account WHERE Industry = 'media'</code>
Relationship query: child-to parent with custom objects	<code>SELECT Id, FirstName__c, Mother_of_Child__r.FirstName__c FROM Daughter__c WHERE Mother_of_Child__r.LastName__c LIKE 'C%'</code>
Relationship query: parent to child with custom objects	<code>SELECT Name, (SELECT Name FROM Line_Items__r) FROM Merchandise__c WHERE Name LIKE 'Acme%'</code>
Relationship queries with polymorphic key	<code>SELECT Id, Owner.Name FROM Task WHERE Owner.FirstName like 'B%'</code> <code>SELECT Id, Who.FirstName, Who.LastName FROM Task WHERE Owner.FirstName LIKE 'B%'</code>

Type of Search	Examples
	<code>SELECT Id, What.Name FROM Event</code>
Polymorphic relationship queries using <code>TYPEOF</code>	<code>SELECT TYPEOF What WHEN Account THEN Phone, NumberOfEmployees WHEN Opportunity THEN Amount, CloseDate ELSE Name, Email END FROM Event</code>
Relationship queries with aggregate	<code>SELECT Name, (SELECT CreatedBy.Name FROM Notes) FROM Account</code> <code>SELECT Amount, Id, Name, (SELECT Quantity, ListPrice, PricebookEntry.UnitPrice, PricebookEntry.Name FROM OpportunityLineItems) FROM Opportunity</code>
Simple query: the UserId and LoginTime for each user	<code>SELECT UserId, LoginTime from LoginHistory</code>
Relationship queries with number of logins per user in a specific time range	<code>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</code>
Query to retrieve the RecordType per user	<code>SELECT Id, Name, IsActive, SubjectType, DeveloperName, Description FROM RecordType</code>
	 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 in your statements. You can use Apex script variables and expressions when preceded by a colon (:).

SOQL SELECT Functions

Use functions in SOQL queries to generate reports for analysis, apply localized formatting to standard and custom fields, group or filter data by date periods, and more.

Functions	Description
Aggregate Functions	Use aggregate functions in a <code>GROUP BY</code> clause in SOQL queries to generate reports for analysis.
<code>convertCurrency()</code>	Use <code>convertCurrency()</code> in SOQL queries to convert currency fields to the user's currency.
<code>convertTimezone()</code>	Use <code>convertTimezone()</code> in a date function of SOQL queries to convert date fields to the user's time zone.
Date Functions	Use date functions in SOQL queries to group or filter data by date periods such as day, calendar month, or fiscal year.
<code>FORMAT()</code>	Use <code>FORMAT()</code> in SOQL queries to apply localized formatting to standard and custom number, date, time, and currency fields.
<code>GROUPING(fieldName)</code>	Use <code>GROUPING(fieldName)</code> in SOQL queries to identify subtotals.

Functions	Description
<code>toLabel()</code>	Use <code>toLabel()</code> to translate SOQL query results into the language of the user who submits the query.

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.

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.



Note: All aggregate functions ignore null values, except for `COUNT()` and `COUNT(Id)`.

`COUNT(fieldname)` is different from `COUNT()`. `COUNT(fieldname)` does ignore null values.

Aggregate Function	Description
<code>AVG()</code>	<p>Returns the average value of a numeric field. For example:</p> <pre>SELECT CampaignId, AVG(Amount) FROM Opportunity GROUP BY CampaignId</pre>
<code>COUNT()</code> and <code>COUNT(fieldName)</code>	<p>Returns the number of rows matching the query criteria. For example using <code>COUNT()</code>:</p> <pre>SELECT COUNT() FROM Account WHERE Name LIKE 'a%'</pre> <p>For example using <code>COUNT(fieldName)</code>:</p> <pre>SELECT COUNT(Id) FROM Account WHERE Name LIKE 'a%'</pre> <p> Note: <code>COUNT()</code> and <code>COUNT(Id)</code> in SOQL are similar to <code>COUNT(*)</code> in SQL.</p>

Aggregate Function	Description
COUNT_DISTINCT()	<p>Returns the number of distinct non-null field values matching the query criteria. For example:</p> <pre>SELECT COUNT_DISTINCT(Company) FROM Lead</pre> <p> Note: COUNT_DISTINCT(<i>fieldName</i>) in SOQL is equivalent to COUNT(DISTINCT <i>fieldName</i>) in SQL.</p>
MIN()	<p>Returns the minimum value of a field. For example:</p> <pre>SELECT MIN(CreatedDate), FirstName, LastName FROM Contact GROUP BY FirstName, LastName</pre> <p>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.</p>
MAX()	<p>Returns the maximum value of a field. For example:</p> <pre>SELECT Name, MAX(BudgetedCost) FROM Campaign GROUP BY Name</pre> <p>Available in API version 18.0 and later.</p>
SUM()	<p>Returns the total sum of a numeric field. For example:</p> <pre>SELECT SUM(Amount) FROM Opportunity WHERE IsClosed = false AND Probability > 60</pre>

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*)

To discover the number of rows that a query returns, use the aggregate function COUNT() in a SELECT statement of a SOQL query.

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 `size` field of the `QueryResult` object returns the number of rows retrieved in the query. The `records` field returns `null`.

Note the following when using `COUNT()`:

- `COUNT()` must be the only element in the `SELECT` list.
- The number of rows returned by `COUNT()` includes null values that match the filtering conditions of the query.
- You can use `COUNT()` with a `LIMIT` clause.
- You can't use `COUNT()` with an `ORDER BY` clause.

COUNT(*fieldName*)

`COUNT(fieldName)` returns the number of rows that match the filtering conditions and have a non-null value for *fieldName*.

For example:

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



Note: `COUNT()` and `COUNT(Id)` in SOQL are similar to `COUNT(*)` in SQL.

For `COUNT(fieldName)`, the 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%'
```

The count is returned in `expr0`, which is the implied alias for the aggregated field.

You can include multiple `COUNT(fieldName)` items in a `SELECT` clause. For example, the following query returns the number of opportunities, and the number of opportunities associated with a campaign.

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

You can use `GROUP BY` clause with `COUNT(fieldName)` 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
```

SEE ALSO:

[Apex Reference Guide: countQuery\(query\)](#)

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
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

Data Type	AVG ()	COUNT ()	COUNT_DISTINCT ()	MIN ()	MAX ()	SUM ()
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 doesn't support grouping. You can't include fields with these field types in a GROUP BY clause.

convertCurrency()

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
```

To format currencies according to the user's locale, 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 Workarounds

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`.

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.

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.

convertTimezone()

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
```

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)
```

 **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 [convertTimezone\(\)](#).

This table lists all the date functions supported by SOQL.

Date Function	Description	Examples
<code>CALENDAR_MONTH()</code>	Returns a number representing the calendar month of a date field.	<ul style="list-style-type: none"> 1 for January 12 for December
<code>CALENDAR_QUARTER()</code>	Returns a number representing the calendar quarter of a date field.	<ul style="list-style-type: none"> 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
<code>CALENDAR_YEAR()</code>	Returns a number representing the calendar year of a date field.	2009
<code>DAY_IN_MONTH()</code>	Returns a number representing the day in the month of a date field.	20 for February 20
<code>DAY_IN_WEEK()</code>	Returns a number representing the day of the week for a date field.	<ul style="list-style-type: none"> 1 for Sunday 7 for Saturday
<code>DAY_IN_YEAR()</code>	Returns a number representing the day in the year for a date field.	32 for February 1
<code>DAY_ONLY()</code>	Returns a date representing the day portion of a <code>dateTime</code> field.	2009-09-22 for September 22, 2009 You can only use <code>DAY_ONLY()</code> with <code>dateTime</code> fields.
<code>FISCAL_MONTH()</code>	Returns a number representing the fiscal month of a date field. This differs from <code>CALENDAR_MONTH()</code> if your organization uses a fiscal year that does not match the Gregorian calendar. This function is not supported if your organization has custom fiscal years enabled.	If your fiscal year starts in March: <ul style="list-style-type: none"> 1 for March 12 for February
<code>FISCAL_QUARTER()</code>	Returns a number representing the fiscal quarter of a date field. This differs from <code>CALENDAR_QUARTER()</code> if your organization uses a fiscal year that does not match the Gregorian calendar.	If your fiscal year starts in July: <ul style="list-style-type: none"> 1 for July 15

Date Function	Description	Examples
	This function is not supported if your organization has custom fiscal years enabled.	<ul style="list-style-type: none"> 4 for June 6
FISCAL_YEAR ()	<p>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.</p> <p>This function is not supported if your organization has custom fiscal years enabled.</p>	2009
HOUR_IN_DAY ()	Returns a number representing the hour in the day for a dateTime field.	<p>18 for a time of 18:23:10</p> <p>You can only use HOUR_IN_DAY() with dateTime fields.</p>
WEEK_IN_MONTH ()	Returns a number representing the week in the month for a date field.	<p>2 for April 10</p> <p>The first week is from the first through the seventh day of the month.</p>
WEEK_IN_YEAR ()	Returns a number representing the week in the year for a date field.	<p>1 for January 3</p> <p>The first week is from January 1 through January 7.</p>

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)
```

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
```

GROUPING (*fieldName*)

You can use the `GROUPING (fieldName)` function in `SELECT`, `HAVING`, and `ORDER BY` clauses.

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.

`GROUPING (fieldName)` returns 1 if the row is a subtotal for the field, and 0 otherwise.

```
SELECT LeadSource, Rating,
    GROUPING(LeadSource) grpLS, GROUPING(Rating) grpRating,
    COUNT(Name) cnt
FROM Lead
GROUP BY ROLLUP(LeadSource, Rating)
```

For more details, see [Identify Subtotals with GROUP BY](#).

toLabel()

Translate SOQL query results into the language of the user who submits the query using the `toLabel` function.



Important: Where possible, we changed noninclusive terms to align with our company value of Equality. We maintained certain terms to avoid any effect on customer implementations.

Any organization can use the `toLabel()` function.

`toLabel` uses the following syntax:

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


Use this function to return translated search results on:

- Regular, multiselect, division, or currency code picklist fields, whose values are returned by the relevant `describe` API call.
- Data category group and data category unique name fields.
- Recordtype names.
- Object History. `HISTORY` is supported on a limited basis. When a tracked field changes, the clause `SELECT toLabel(Field) FROM *History` returns the primary label of the changed field or the existing translated label. `OldValue` and `NewValue` fields in history objects also support the `toLabel` method, but the query returns translated values for only tracked changes on supported field types.

For example:

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

The returned records are translated into the language of 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.

`toLabel()` supports aliasing, which is required when the query includes the same field multiple times. For example:

```
SELECT Company, Status, toLabel(Status) translatedStatus FROM Lead
```

You can use the `toLabel()` method to filter records using a translated picklist value. For example:

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

The query returns lead records with a picklist value for Status equal to 'le Draft.' The comparison is made against the value for the user's language. If no translation is available for the picklist, the comparison is made against the master values.

The following limitations apply to translated picklist values:

- The `LIKE` operator can only query on the label of the picklist, not on its API name.
- You can't use the `toLabel()` function with the operator `ORDER BY`. SOQL always uses the picklist's defined order.
- You can't use the `toLabel()` function in a `WHERE` clause for division or currency ISO code picklists.

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 SOQL 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](#)
- [null Values in 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 query, 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 `describeObjects()`, 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 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.

Query Child-to-Parent Relationships

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

```
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 Relationships

Query parent-to-child relationships, which are almost always one-to-many, using subqueries enclosed in parentheses. Specify these relationships in a subquery, where the initial member of the `FROM` clause in the subquery is related to the initial member of the outer query `FROM` clause. For standard object subqueries, the name of the relationship is the plural name of the child object.

In API version 58.0 and later, SOQL queries can contain up to five levels of parent-to-child relationships. The ability to query five levels of parent-child relationships is limited to SOQL queries via the REST and SOAP query calls for standards and custom objects.

In each relationship, the parent is counted as the first level of the query and the child relationships can be up to four levels deep from the parent root. SOQL queries that include more than five levels of parent-to-child relationships result in an error.

This example SOQL query includes five levels of parent-child relationships.

```
SELECT Name,
  (SELECT LastName,
    (SELECT AssetLevel,
      (SELECT Description,
        (SELECT LineItemNumber FROM WorkOrderLineItems)
      FROM WorkOrders)
    FROM Assets)
  FROM Contacts)
FROM Account
```

If you include a `WHERE` clause in a subquery, the clause filters on any object in the current scope that is reachable from the parent root of the query, via the parent relationships.

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

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

To understand the versioned behavior of SOQL queries that traverse parent-to-child relationships, see [Relationship Query Limitations](#).

Traverse Relationship Queries

Here are some examples of relationship queries that traverse both parent-to-child and child-to-parent relationships.

This example query returns the name of all the accounts in an organization, and for each account, the name of the user who created each note for the account. If there were no notes for the account, the result set is empty.

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

Another example query that traverses parent-to-child and child-to-parent relationships.

```
SELECT Amount, Id, Name, (SELECT Quantity, ListPrice,
  PriceBookEntry.UnitPrice, PricebookEntry.Name,
  PricebookEntry.product2.Family FROM OpportunityLineItems)
FROM Opportunity
```

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:

Daughter

New Relationship

[Help for this Page](#) ?

Step 3. Enter the label and name for the lookup field
Step 3 of 6

[Previous](#) [Next](#) [Cancel](#)

Field Label ⓘ

Field Name ⓘ

Description

Help Text

ⓘ

Child Relationship Name ⓘ

Required ☒ Always require a value in this field in order to save a record

What to do if the lookup record is deleted?

☐ Clear the value of this field. You can't choose this option if you make this field required.

☐ Delete this record also. This allows users to delete large numbers of records without regard for sharing or visibility constraints.

☒ Don't allow deletion of the lookup record that's part of a lookup 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):

Daughter Custom Field

Mother of Child

[Back to Daughter](#)[Help for this Page](#) ?[Validation Rules](#) [0]Custom Field Definition
Detail[Edit](#)[Set Field-Level Security](#)[View Field Accessibility](#)

Field Information

Field Label	Mother of Child	Object Name	<u>Daughter</u>
Field Name	Mother_of_Child	Data Type	Lookup
API Name	Mother_of_Child__c		
Description	Relationship Mother of Child		
Help Text			
Created By	<u>Peter Conrad</u> , 9/19/2014 12:36 PM	Modified By	<u>Peter Conrad</u> , 9/19/2014 12:36 PM

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 query 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.

The following sample illustrates how to process subquery results:

```
private void querySample() {
    QueryResult qr = null;
```



```

try {
    qr = connection.query("SELECT a.Id, a.Name, " +
        "(SELECT c.Id, c.FirstName, " +
        "c.LastName FROM a.Contacts c) FROM Account a");
    boolean done = false;
    if (qr.getSize() > 0) {
        while (!done) {
            for (int i = 0; i < qr.getRecords().length; i++) {
                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 (qr.getSize() > 0) {
        while (!done) {
            for (int i = 0; i < qr.getRecords().length; i++) {
                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 {
                qr = connection.queryMore(qr.getQueryLocator());
            }
        }
    } else {
        System.out.println("No child contacts found.");
    }
}
}

```

null Values in Lookup Relationships and Outer Joins

Relationship SOQL queries return records, even if the relevant foreign key field has a null value, as with an outer join.

- In an `ORDER BY` clause, the record is returned even if the foreign key value in a record is null. 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.

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. Records are returned even if `Parent__r.id` or `Parent__r.name` are null.

- In a `WHERE` clause that uses `OR`, records are returned even if the foreign key value in a record is null. For example, if your organization has one contact with the value of its `LastName` field equal to `Young` and the value of its `AccountId` field equal to `null`, and another contact with a different last name and a parent account named `Quarry`, the following query returns both contacts:

```
SELECT Id FROM Contact WHERE LastName = 'Young' or Account.Name = 'Quarry'
```

- In a `WHERE` clause that checks for a value in a parent field, the record is returned even if the parent does not exist.. For example:

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

Case record `Id` values are returned.

- 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 query.

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`:

```
<complexType name="Account">
  <complexContent>
    <extension base="ens:sObject">
      <sequence>
        ...
        <element name="Contacts" nillable="true" minOccurs="0"
```

```

                                type="tns:QueryResult"/>
                                ...
                            </sequence>
                        </extension>
                    </complexContent>
                </complexType>

```

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"`.

```

<complexType name="Opportunity">
  <complexContent>
    <extension base="ens:sObject">
      <sequence>
        ...
        <element name="Account" nillable="true" minOccurs="0"
          type="ens:Account"/>
        <element name="AccountId" nillable="true" minOccurs="0"
          type="tns:ID"/>
        ...
      </sequence>
    </extension>
  </complexContent>
</complexType>

```



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`:

```

<complexType name="Mother__c">
  <complexContent>
    <extension base="ens:sObject">
      <sequence>
        ...
        <element name="Daughters__r" nillable="true" minOccurs="0"
          type="tns:QueryResult"/>
        <element name="FirstName__c" nillable="true" minOccurs="0"
          type="xsd:string"/>
        <element name="LastName__c" nillable="true" minOccurs="0"
          type="xsd:string"/>
        ...
      </sequence>
    </extension>
  </complexContent>
</complexType>

```

```

<complexType name="Daughter__c">
  <complexContent>
    <extension base="ens:sObject">
      <sequence>
        ...

```

```

<element name="Mother_of_Child__c" nillable="true" minOccurs="0"
  type="tns:ID"/>
<element name="Mother_of_Child__r" nillable="true" minOccurs="0"
  type="xsd:string"/>
<element name="LastName__c" nillable="true" minOccurs="0"
  type="ens:Mother__c"/>
...
</sequence>
</extension>
</complexContent>
</complexType>

```

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 `describeObjects()` 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, `polymorphicForeignKey` is true, and `referenceTo` has more than one referenced object type, 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 Account
```



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
- `polymorphicForeignKey` = 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` can't be used with a relationship field whose `relationshipName` 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. For details, see [Filter on Polymorphic Relationship Fields](#).
- `TYPEOF` isn't allowed in queries that don't return objects, such as `COUNT ()`.
- `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 because TYPEOF is used in the semi-join query:

```
SELECT Name FROM Account
WHERE CreatedById IN
(
  SELECT
    TYPEOF Owner
      WHEN User THEN Id
      WHEN Group THEN CreatedById
    END
  FROM CASE
)
```

The following example 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
)
```

- TYPEOF can't be used in queries with functions in the SELECT clause. The following example is not valid because the TYPEOF includes the FORMAT function.

```
SELECT
  TYPEOF What
    WHEN Account THEN Id, FORMAT(LastModifiedDate) LastModifiedDate__f
    WHEN Oppty THEN Id
  END
FROM Task
```

Instead, run the same query without functions to retrieve a list of IDs.

```
SELECT
  TYPEOF What
    WHEN Account THEN Id, LastModifiedDate
    WHEN Opportunity THEN Id
  END
FROM Task
```

Then, run a second query with functions on the resulting ID list.

```
SELECT
  FORMAT(LastModifiedDate) LastModifiedDate__f
FROM Account
WHERE Id in RetrievedIdList
```

- TYPEOF can't be used in queries with [GROUP BY](#), [GROUP BY ROLLUP](#), [GROUP BY CUBE](#), and [HAVING](#).

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:

```
<complexType name="Task">
  <complexContent>
    <extension base="ens:sObject">
      <sequence>
        ...
        <element name="Owner" nillable="true" minOccurs="0" type="ens:sObject"/>
        ...
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

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

```
<complexType name="Task">
  <complexContent>
    <extension base="ens:sObject">
      <sequence>
        ...
        <element name="Owner" nillable="true" minOccurs="0" type="ens:Name"/>
        ...
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

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](#).

SEE ALSO:

[Apex Reference Guide: Working with Polymorphic Relationships in SOQL Queries](#)

[Apex Reference Guide: getObject\(field\)](#)

Understanding Relationship Query Limitations

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

- Relationship queries aren't the same as SQL joins. You must have a relationship between objects to create a join in SOQL.
- No more than 55 child-to-parent relationships can be specified in a query. A custom object allows up to 40 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 query.
- 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 API version 57.0 and earlier, only two levels of parent-to-child relationship can be specified in a query.
- In API version 58.0 and later, up to five levels of parent-to-child relationship can be queried via REST and SOAP query calls for standard and custom objects. SOQL queries with five level parent-to-child relationships aren't supported for big objects, external objects, or in Apex, Bulk API, and Bulk API 2.0.
- 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
```

- You can't use the `USING SCOPE` clause to limit the results of a parent-child relationship query to a specified scope.
- Consider these limitations for external objects.
 - A subquery that involves external objects or a filter on parent 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 query 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, `whoId`, `whatId`, or `ownerId` 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](https://developer.salesforce.com) (requires login).

Change the Batch Size in Queries

The batch size for a query determines the number of rows that are returned in the query results. You can change the batch size that's returned in queries made using both REST API and SOAP API.

The default and the maximum for query result batch size is 2,000 records. However, to optimize performance, the returned batch size can be lower than the max or what's set in the request, based on the size and complexity of records queried.

In REST API, use the `batchSize` field in the [query options header](#) to change the number of results returned in a single query. For more information, see [Query](#) in the *REST API Developer Guide*.

In SOAP API, change the batch size in the `QueryOptions` header before invoking the `query()` call. Salesforce Web Service Connector (WSC) clients can set the batch size by calling `setQueryOptions()` on the connection object. For more information, see [query\(\)](#) in the *SOAP API Developer Guide*.



Note: If the SOQL statement selects two or more custom fields of type long text, the batch size can't be greater than 200 records. This limit prevents large SOAP messages from being returned.

WSC (Java) Example to Set the Batch Size

The following Java WSC (Java) code example sets the batch size to 250 records.

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

C# (.NET) Example to Set the Batch Size

The following C# (.NET) code example sets the batch size to 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
<code>ContentDocumentLink</code>	A SOQL query must filter on one of <code>Id</code> , <code>ContentDocumentId</code> , or <code>LinkedEntityId</code> .
<code>ContentHubItem</code>	A SOQL query must filter on one of <code>Id</code> , <code>ExternalId</code> , or <code>ContentHubRepositoryId</code> .
Custom metadata types	Custom metadata types support the following SOQL query syntax. <div data-bbox="555 1743 976 1843" data-label="Text"> <pre>SELECT fieldList [...] FROM objectType [USING SCOPE filterScope]</pre> </div>

Object	Description
	<div> <pre>[WHERE conditionExpression] [ORDER BY field {ASC DESC} [NULLS {FIRST LAST}}]</pre> </div> <ul style="list-style-type: none"> You can use metadata relationship fields in the <i>fieldList</i> and <i>conditionExpression</i>. FROM can include only 1 object. You can use the following operators. <ul style="list-style-type: none"> IN and NOT IN =, >, >=, <, <=, and != LIKE, including wild cards AND OR when on the same column with LIKE and = operations <p> Note: OR can't be used as a compound filter when child filters are on two different columns.</p> <ul style="list-style-type: none"> You can use ORDER BY only with non-relationship fields. You can use ORDER BY, ASC, and DESC with multiple (non-relationship) fields. Metadata relationship fields support all standard relationship queries.
Big objects	<ul style="list-style-type: none"> 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. <ul style="list-style-type: none"> =, <, >, <=, >=, and IN Any prior fields in a query can use only the = operator. Big objects don't support the following operators. <ul style="list-style-type: none"> !=, LIKE, NOT IN, EXCLUDES, and INCLUDES
External objects	<ul style="list-style-type: none"> 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. <ul style="list-style-type: none"> AVG () function COUNT (fieldName) function (however, COUNT () is supported) HAVING clause GROUP BY clause MAX () function MIN () function SUM () function

Object	Description
	<ul style="list-style-type: none"> External objects also don't support the following. <ul style="list-style-type: none"> EXCLUDES operator FOR VIEW clause FOR REFERENCE clause INCLUDES operator LIKE operator toLabel () function TYPEOF clause WITH clause <p>The following limits apply only to the OData 2.0 and 4.0 adapters for Salesforce Connect.</p> <ul style="list-style-type: none"> External objects have the following limitations for the ORDER BY clause. <ul style="list-style-type: none"> 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. <p>The following limits apply only to custom adapters for Salesforce Connect.</p> <ul style="list-style-type: none"> Location-based SOQL queries of external objects aren't supported. If a SOQL query of an external object includes the following, the query fails. <ul style="list-style-type: none"> convertCurrency () function UPDATE TRACKING clause UPDATE VIEWSTAT clause USING SCOPE clause In an ORDER BY clause, the following are ignored. <ul style="list-style-type: none"> 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.
KnowledgeArticleVersion	<ul style="list-style-type: none"> Filter on a single value of PublishStatus for best results. To find all versions of each article, omit the PublishStatus filter, but do filter on one or more primary key IDs. To retrieve all archived versions for a given article, specify a SOQL filter where IsLatestVersion is false. In API version 46.0 and earlier, queries without a filter on PublishStatus return published articles by default. In API version 47.0 and later, draft, published, and archived articles are returned when Lightning Knowledge is enabled.

Object	Description
	<ul style="list-style-type: none"> To support security, only users with the “View Draft Articles” permission see articles whose <code>PublishStatus</code> value is <code>Draft</code>. Similarly, only users with the “View Archived Articles” permission see articles whose <code>PublishStatus</code> value is <code>Archived</code>. Archived article versions are stored in the Knowledge__kav object. To query archived article versions, specify the article <code>Id</code> and set <code>IsLatestVersion='0'</code>. You can’t use binding variables in Apex SOQL statements with <code>KnowledgeArticleVersion</code> objects. For example, the following SOQL statement causes a compilation error. <pre>final String PUBLISH_STATUS_ONLINE = 'Online'; List<Knowledge__kav> articles = [SELECT Id FROM Knowledge__kav WHERE PublishStatus = :PUBLISH_STATUS_ONLINE];</pre> <p>Instead, use dynamic SOQL as follows. See Dynamic SOQL in <i>Apex Developer Guide</i>.</p> <pre>final String PUBLISH_STATUS_ONLINE = 'Online'; final String q = 'SELECT Id, PublishStatus FROM Knowledge__kav WHERE PublishStatus = :PUBLISH_STATUS_ONLINE'; List<Knowledge__kav> articles = Database.query(q);</pre>
NewsFeed	<ul style="list-style-type: none"> No SOQL limit if logged-in user has “View All Data” permission. If not, specify a <code>LIMIT</code> clause of 1,000 records or fewer. SOQL <code>ORDER BY</code> on fields using relationships is not available. Use <code>ORDER BY</code> on fields on the root object in the SOQL query.
RecentlyViewed	The <code>RecentlyViewed</code> object is updated every time the logged-in user views or references a record. It is also updated when records are retrieved using the <code>FOR VIEW</code> or <code>FOR REFERENCE</code> clause in a SOQL query. To ensure that the most recent data is available, <code>RecentlyViewed</code> data is periodically truncated down to 200 records per object. <code>RecentlyViewed</code> data is retained for 90 days, after which it is removed on a periodic basis.
TopicAssignment	<p>No SOQL limit if logged-in user has “View All Data” permission. If not, do one of the following:</p> <ul style="list-style-type: none"> Specify a <code>LIMIT</code> clause of 1,100 records or fewer. Filter on <code>Id</code> or <code>Entity</code> when using a <code>WHERE</code> clause with <code>=</code>.
UserRecordAccess	<ul style="list-style-type: none"> Always use the query formats specified in the SOAP API Developer Guide. May include an <code>ORDER BY</code> clause. You must <code>ORDER BY HasAccess</code> if you <code>SELECT HasAccess</code>, and <code>ORDER BY MaxAccessLevel</code> if you <code>SELECT MaxAccessLevel</code>. Maximum number of records that can be queried is 200.
UserProfileFeed	<ul style="list-style-type: none"> No SOQL limit if logged-in user has “View All Data” permission. If not, specify a <code>LIMIT</code> clause of 1,000 records or fewer.

Object	Description
	<ul style="list-style-type: none"> SOQL ORDER BY on fields using relationships is not available. Use ORDER BY on fields on the root object in the SOQL query. <p>Also, a SOQL query must include <code>WITH UserId = {userId}</code>.</p>
Vote	<ul style="list-style-type: none"> <code>ParentId = [single ID]</code> <code>Parent.Type = [single type]</code> <code>Id = [single ID]</code> <code>Id IN = [list of IDs]</code>

For other SOQL limits, see [SOQL and SOSL Limits](#) in the *Salesforce Developer Limits and Allocations Quick Reference*.

SOQL with Big Objects

You can query the fields in a big object's index by 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 `IN` on any field in your query, although you can use `IN` only one time. You can use the range operations `<`, `>`, `<=`, or `>=` only on the last field of your query.



Tip: When you use the `IN` clause with only one argument, such as `FirstName IN ('Charlie')`, it's equivalent to using `=`, such as `FirstName='Charlie'`. For clarity, we suggest you use the `=` form in this case.

Subqueries aren't supported. Don't include more than one select statement in your query. For example, this query is not supported

```
Select CreatedById, CreatedDate, Created_Date__c, Id, Legacy_Record_ID__c, Parent_Case__c,
  SystemModstamp, Text_Body__c FROM Archived_Email_Message__b WHERE Parent_Case__c IN(select
  id from case where owner.id in ('00580000008BBVUAA4'))
```

You can include the system fields `CreatedById`, `CreatedDate`, and `SystemModstamp` in queries.

To guarantee the order of the query results, use the `ORDER BY` clause.

These queries assume that you have a table where 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 use a range operator.

```
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 use a range operator.

```
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 use a range operator.

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

This query uses the `IN` operator on the first field in the index.

```
SELECT LastName__c, FirstName__c, PhoneNumber__c
FROM Phone_Book__b
WHERE LastName__c IN ('Kelly','Jones','Capulet','Montague') AND FirstName__c='Charlie'
```

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'
```

This query also doesn't work because it uses the `IN` operator twice.

```
SELECT LastName__c, FirstName__c, PhoneNumber__c
FROM Phone_Book__b
WHERE LastName__c IN ('Kelly','Jones') AND FirstName__c IN ('Charlie','Lisa')
```

This query works, even though it appears to have two `IN` operators in the `WHERE` clause. But because the second `IN` has only one argument, it's equivalent to an equals operator, so it's allowed.

```
SELECT LastName__c, FirstName__c, PhoneNumber__c
FROM Phone_Book__b
WHERE LastName__c IN ('Kelly','Jones') AND FirstName__c IN ('Charlie')
```

For clarity, we suggest that you rewrite the preceding SOQL statement as shown.

```
SELECT LastName__c, FirstName__c, PhoneNumber__c
FROM Phone_Book__b
WHERE LastName__c IN ('Kelly','Jones') AND FirstName__c='Charlie'
```

SOQL Operations Not Allowed with Big Objects

- When building an index query, don't leave gaps between the first and the last field in the query.
- The operators `!=`, `LIKE`, `NOT IN`, `EXCLUDES`, and `INCLUDES` aren't valid in any query.
- Aggregate functions aren't valid in any query.
- To retrieve a list of results, don't use the `Id` field in a query. Including `Id` in a query returns only results that have an empty ID (0000000000000000 or 0000000000000000AAA).



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.

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 *Salesforce Object Reference*.

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/objects/Warehouse__c/a06D0000001704nIAE"
    },
    "Name" : "Ferry Building Depot",
    "Location__c" : {
```

```

        "latitude" : 37.79302,
        "longitude" : -122.394507
    }, {
        "attributes" : {
            "type" : "Warehouse__c",
            "url" : "/services/data/v30.0/objects/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).



Note: If you use a location field and a `GEOLOCATION` value, the location field must be the first variable and the `GEOLOCATION` must be the second variable. If the `GEOLOCATION` is first, the query results in a `MALFORMED_QUERY` error.

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

```

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	Location__c
Ferry Building Depot	<i>null</i>
Aloha Warehouse	(37.786108,-122.430152)
Big Tech Warehouse	<i>null</i>
S H Frank & Company	<i>null</i>
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	Location__latitude__s	Location__longitude__s
Ferry Building Depot	<i>null</i>	-122.394507
Aloha Warehouse	37.786108	-122.430152
Big Tech Warehouse	<i>null</i>	<i>null</i>
S H Frank & Company	37.763662	<i>null</i>
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 the `DISTANCE` function. 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];
```

For more information, see [Compound Field Considerations and Limitations](#) in the *Salesforce Object Reference*.

CHAPTER 3 Salesforce Object Search Language (SOSL)

In this chapter ...

- [Typographical Conventions in This Document](#)
- [SOSL Limits on Search Results](#)
- [SOSL Limits on External Object Search Results](#)
- [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](#)
- [WITH DATA CATEGORY DataCategorySpec](#)
- [WITH DivisionFilter](#)
- [WITH HIGHLIGHT](#)
- [WITH METADATA](#)
- [WITH NETWORK NetworkIdSpec](#)
- [WITH PricebookId](#)
- [WITH SNIPPET](#)

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

When building efficient SOSL queries, create filters that are selective. Search analyzes all records in the index and by default, only the top 2,000 matched records are returned based on relevancy. This API supports pagination to fetch additional matches. Sharing is applied after the result set is returned from the search stack. If your filters aren't selective and cause search term matches of more than 2000 records, there's a possibility of running into search crowding.

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

- SOAP API search() calls. See [search\(\)](#) in the *SOAP API Developer Guide*.
- REST API Search calls. See [Search](#) in the *REST API Developer Guide*.
- Apex statements. See [SOQL and SOSL Queries](#) in the *Apex Developer Guide*.
- Visualforce controllers and getter methods. See [Controller Methods](#) in the *Visualforce Developer Guide*.
- Schema Explorer of the Eclipse Toolkit. See [Schema Explorer](#) in the *Force.com IDE Developer Guide*.

 **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're 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 SPELL_CORRECTION](#)
- `WITH DATA CATEGORY`: Specifies the data categories to return.
- `WITH DivisionFilter`: Specifies the division field to return.
- `WITH NETWORK`: Specifies the Experience Cloud site ID to return.
- `WITH PricebookId`: Specifies the price book ID to return.

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
<code>FIND Name IN Account</code>	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.
<code>FIND <i>fieldname</i> IN <i>objectname</i></code>	Italics represent a variable or placeholder. You supply the actual value.
<code> </code>	The pipe character separates alternate elements. For example, in the clause <code>UPDATE TRACKING VIEWSTAT [, ...]</code> , the <code> </code> character indicates that you can use either <code>TRACKING</code> or <code>VIEWSTAT</code> after <code>UPDATE</code> .
<code>[LIMIT n]</code>	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 <i>fieldname</i> [ASC DESC] [NULLS {FIRST LAST}]] ,ASC, DESC, or the NULLS clause cannot be used without the ORDER BY clause.</code>
<code>[...] and [, ...]</code>	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 <code>[[[UPDATE [TRACKING VIEWSTAT] [, ...]]]]</code> , the <code>[, ...]</code> indicates that you can use <code>TRACKING</code> , <code>VIEWSTAT</code> , or both: <div data-bbox="646 1234 1446 1409" style="border: 1px solid #add8e6; padding: 5px; margin-top: 10px;"> <code>UPDATE TRACKING</code> </div> <div data-bbox="646 1297 1446 1346" style="border: 1px solid #add8e6; padding: 5px; margin-top: 5px;"> <code>UPDATE VIEWSTAT</code> </div> <div data-bbox="646 1360 1446 1409" style="border: 1px solid #add8e6; padding: 5px; margin-top: 5px;"> <code>UPDATE TRACKING, VIEWSTAT</code> </div>

SOSL Limits on Search Results

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.

For other SOSL limits, see [SOQL](#) and [SOSL Limits](#) in the *Salesforce Developer Limits and Allocations Quick Reference*.

SOSL Limits on External Object Search Results

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] ]
```


 **Note:** `OFFSET n` and `WHERE` are included within `RETURNING FieldSpec`.

where:

Syntax	Description
<code>convertCurrency()</code>	Optional. If an org has multicurrency enabled, converts currency fields to the user's currency.
<code>FIND {SearchQuery}</code>	<p>Required. Specifies the text (words or phrases) to search for. Enclose the search query with curly braces.</p> <p>If the <code>SearchQuery</code> string is longer than 10,000 characters, no result rows are returned. If <code>SearchQuery</code> is longer than 4,000 characters, any logical operators are removed. For example, the <code>AND</code> operator in a statement with a <code>SearchQuery</code> that's 4,001 characters will default to the <code>OR</code> operator, which could return more results than expected.</p>
<code>FORMAT ()</code>	Optional. Use <code>FORMAT</code> with the <code>FIND</code> clause to apply localized formatting to standard and custom number, date, time, and currency fields. The <code>FORMAT</code> function supports aliasing. In addition, aliasing is required when the query includes the same field multiple times.
<code>IN SearchGroup</code>	<p>Optional. Scope of fields to search. One of the following values:</p> <ul style="list-style-type: none"> • <code>ALL FIELDS</code> • <code>NAME FIELDS</code> • <code>EMAIL FIELDS</code> • <code>PHONE FIELDS</code> • <code>SIDEBAR FIELDS</code> <p>If unspecified, the default isn't <code>ALL FIELDS</code>. You can specify the list of objects to search in the <code>RETURNING FieldSpec</code> clause.</p> <p> Note: This clause doesn't apply to articles, documents, feed comments, feed items, files, products, and solutions. If these objects are specified in the <code>RETURNING</code> clause, the search is limited to specific fields, and all fields are searched.</p>
<code>LIMIT n</code>	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
<code>OFFSET n</code>	Optional. When expecting many records in a query's results, you can display the results in multiple pages by using the <code>OFFSET</code> clause in a SOSL query. For example, you can use <code>OFFSET</code> to display records 51–75 and then jump to displaying records 301–350. Using <code>OFFSET</code> is an efficient way to handle large results sets.
<code>ORDER BY</code>	Optional. Specifies the order in which search results are returned using the <code>ORDER BY</code> clause. You can also use this clause to display empty records at the beginning or end of the results.
<code>RETURNING <i>FieldSpec</i></code>	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.
<code>toLabel(fields)</code>	Optional. Results from a query are returned translated into the user's language.
<code>USING ListView=</code>	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.
<code>[UPDATE [TRACKING VIEWSTAT][,...]]</code>	Optional. If an org uses Salesforce Knowledge: <ul style="list-style-type: none"> • <code>UPDATE TRACKING</code> tracks keywords used in Salesforce Knowledge article search. • <code>Update an Article's Viewstat with SOSL</code> updates an article's view statistics. • <code>UPDATE TRACKING, VIEWSTAT</code> does both.
<code>WHERE</code>	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.
<code>WITH DATA CATEGORY <i>DataCategorySpec</i></code>	Optional. If an org uses Salesforce Knowledge articles or answers, filters all search results based on one or more data categories.
<code>WITH <i>DivisionFilter</i></code>	Optional. If an org uses divisions, filters all search results based on values for the <code>Division</code> field.
<code>WITH HIGHLIGHT</code>	Optional. In search results for certain objects, highlights the terms matching the search query.
<code>WITH METADATA = MetadataSpec</code>	Optional. Specifies if metadata is returned in the response. The default setting is no, meaning no metadata is returned.
<code>WITH NETWORK NetworkIdSpec</code>	Optional. Filters search results by community ID.
<code>WITH PricebookId</code>	Optional. Filters product search results by a single price book ID.

Syntax	Description
<code>WITH SNIPPET [(target_length=n)]</code>	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 <code>target_length</code> parameter to specify an alternate target length, which can be from 100 and 500 characters.
<code>WITH SPELL_CORRECTION</code>	Optional. When set to <code>true</code> , spell correction is enabled for searches that support spell correction. When set to <code>false</code> , spell correction isn't enabled. The default value is <code>true</code> .

 **Note:** The SOSL statement character limit is tied to the SOQL statement character limit defined for your org. By default, SOQL and SOSL queries can't exceed 100,000 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. 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 a people record is called Joe Smith or Joe Smythe, that record isn't 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 &|!(){}[]^"~*?:\'+-

```
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 in your statements. 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. So, `convertCurrency()` can't be used with the [ORDER BY Clause](#).

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

```
FIND {Acme} RETURNING Account (AnnualRevenue, convertCurrency(AnnualRevenue) AliasCurrency)
```

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 Tokyo Prefecture	京都 Kyoto

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

東京 Tokyo	都 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
*	<p>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 meyers</i> or <i>michael meyers</i>.</p> <p>If you are searching for a literal asterisk in a word or phrase, then escape the asterisk (precede it with the \ character).</p>
?	<p>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.</p>

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
" "	<p>Use quotation marks around search terms to find matches in the order you entered your search terms. A search for <code>"monday meeting"</code> finds items that contain <code>monday meeting</code> in that order.</p> <p>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.</p>

Operator	Description
AND	Finds items that match all the search terms. For example, <code>john AND smith</code> finds items with both the word <code>john</code> and the word <code>smith</code> . 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, <code>john OR smith</code> finds items with either <code>john</code> or <code>smith</code> , or both words.
AND NOT	Finds items that do not contain the search term. For example, <code>john AND NOT smith</code> finds items that have the word <code>john</code> but not the word <code>smith</code> .
()	Use parentheses around search terms with logical operators to group search terms. For example, you can search for: <ul style="list-style-type: none"> <code>("Bob" and "Jones") OR ("Sally" and "Smith")</code>—searches for either Bob Jones or Sally Smith. <code>("Bob") and ("Jones" OR "Thomas") and Sally Smith</code>—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 defaults 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. OR

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

FIND Clauses in Apex

The syntax of the `FIND` clause in Apex differs from the syntax of the `FIND` clause in 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
```



Note: Apex that is running in system mode ignores field-level security while scanning for a match using `IN ALL FIELDS`.

- 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 *Apex Developer 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

In an SOSL query, you can specify which types of text fields to search for using the `IN SearchGroup` optional clause. The `SearchGroup` defines the scope of the search. For example, you can search for name, email, phone, sidebar, or all fields.

The following table lists the valid `SearchGroup` values. If you don't specify a value, the default behavior is to search all text fields in searchable objects. Numeric fields aren't searchable.



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

SearchGroup	Description
ALL FIELDS	Search all searchable fields. If the <code>IN</code> clause is unspecified, then <code>ALL FIELDS</code> is the default setting.
EMAIL FIELDS	Search only email fields.
NAME FIELDS	<p>Search only name fields for standard objects.</p> <p>In addition to the Name field, Salesforce searches the following fields when using <code>IN NAME FIELDS</code> with these standard objects:</p> <ul style="list-style-type: none"> Account: Website, Site, NameLocal Asset: SerialNumber Case: SuppliedName, SuppliedCompany, Subject Contact: AssistantName, FirstNameLocal, LastNameLocal, AccountName Event: Subject Lead: Company, CompanyLocal, FirstNameLocal, LastNameLocal Note: Title PermissionSet: Label Report: Description

SearchGroup	Description
	<ul style="list-style-type: none"> TagDefinition: NormName Task: Subject User: CommunityNickname <p>In custom objects, fields that are defined as “Name Field” are searched. In standard and custom objects, name fields have the <code>nameField</code> property set to <code>true</code>. (See the <code>Field</code> array of the <code>fields</code> parameter of the <code>DescribeSObjectResult</code> for more information.)</p>
PHONE_FIELDS	Search only phone number fields.
SIDEBAR_FIELDS	Search for valid records as listed in the Sidebar dropdown list. Unlike search in the application, the asterisk (*) wildcard isn’t appended to the end of a search string.

The `IN` clause is optional, but it limits the search scope, making the search more efficient. For example, to search only for an email address, specify `IN EMAIL_FIELDS`.

Examples of IN Clauses

Search Type	Examples
No search group	<code>FIND {MyProspect}</code>
ALL_FIELDS	<code>FIND {MyProspect} IN ALL_FIELDS</code>
EMAIL_FIELDS	<code>FIND {mylogin@mycompany.com} IN EMAIL_FIELDS</code>
NAME_FIELDS	<code>FIND {MyProspect} IN NAME_FIELDS</code>
PHONE_FIELDS	<code>FIND {MyProspect} IN PHONE_FIELDS</code>
SIDEBAR_FIELDS	<code>FIND {MyProspect} IN SIDEBAR_FIELDS</code>
Invalid search	<code>FIND {MyProspect} IN Accounts</code>

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: SOSL 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

```
ORDER BY fieldname [ASC | DESC] [NULLS [first | last]]
```

Syntax	Description
ASC or DESC	Orders the results in ascending or descending order. The default is ascending. You can have more than one <code>ORDER BY</code> clause.
NULLS [<i>first</i> <i>last</i>]	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)
```

RETURNING *FieldSpec*

`RETURNING` is an optional clause that you can add to a SOSL query to specify the information to be returned in the text search result.

If you don't specify this clause, the default behavior is to return the IDs of all searchable objects in advanced search up to the maximum limit. The maximum limit is specified in the [LIMITn](#) clause or 2,000 (API version 28.0 and later), whichever is smaller. The returned IDs include custom objects even if there's no custom tab. Search results list objects in the order specified in the clause. 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] [USING Listview=listview name] [ORDER BY Clause] [LIMIT n] [OFFSETn])]
[, ObjectTypeName [(FieldList [WHERE] [ORDER BY Clause] [LIMITn] [OFFSETn])]]
```

RETURNING can contain the following elements:

Name	Description
<i>ObjectTypeName</i>	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 <i>ObjectTypeName</i> , each object must be distinct; you can't repeat an <i>ObjectTypeName</i> within a single RETURNING clause. The search () call only returns objects specified in the RETURNING clause.
<i>FieldList</i>	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.
<i>USING ListView=</i>	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. <i>ListView=Recent</i> searches for the most recently accessed items viewed or referenced by the current user.
<i>WHERE</i>	<p>Optional description of how search results for the given object are filtered, based on individual field values. If unspecified, the search retrieves all the rows in the object that are visible to the user.</p> <p>If you want to specify a WHERE clause, you must include a <i>FieldList</i> with at least one specified field. The following example isn't proper syntax:</p> <pre>RETURNING Account (WHERE name like 'test')</pre> <p>But this syntax is:</p> <pre>RETURNING Account (Name, Industry WHERE Name like 'test')</pre> <p>See conditionExpression for more information.</p>

Name	Description
<code>ORDER BY Clause</code>	<p>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 <code>ORDER BY</code> clause.</p> <p>If you want to specify an <code>ORDER BY</code> clause, you must include a <i>FieldList</i> with at least one specified field. The following example isn't proper syntax:</p> <pre>RETURNING Account (ORDER BY id)</pre> <p>But this syntax is:</p> <pre>RETURNING Account (Name, Industry ORDER BY Name)</pre>
<code>LIMITn</code>	<p>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.</p> <p>If you want to specify a <code>LIMIT</code> clause, you must include a <i>FieldList</i> with at least one specified field. The following example isn't proper syntax:</p> <pre>RETURNING Account (LIMIT 10)</pre> <p>But this syntax is:</p> <pre>RETURNING Account (Name, Industry LIMIT 10)</pre>
<code>OFFSETn</code>	<p>Optional clause used to specify the starting row offset into the result set returned by your query. <code>OFFSET</code> can be used only when querying a single object. <code>OFFSET</code> must be the last clause specified in a query.</p> <p>If you want to specify an <code>OFFSET</code> clause, you must include a <i>FieldList</i> with at least one specified field. The following example isn't proper syntax:</p> <pre>RETURNING Account (OFFSET 25)</pre> <p>But this syntax is:</p> <pre>RETURNING Account (Name, Industry OFFSET 25)</pre>



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	Examples
No Field Spec	<code>FIND {MyProspect}</code>
One sObject, no fields	<code>FIND {MyProspect} RETURNING Contact</code>
Multiple sObject objects, no fields	<code>FIND {MyProspect} RETURNING Contact, Lead</code>
One sObject, one or more fields	<code>FIND {MyProspect} RETURNING Account (Name)</code>

Search Type	Examples
	<code>FIND {MyProspect} RETURNING Contact(FirstName, LastName)</code>
Custom sObject	<code>FIND {MyProspect} RETURNING CustomObject_c</code>
	<code>FIND {MyProspect} RETURNING CustomObject_c(CustomField_c)</code>
Multiple sObject objects, one or more fields, limits	<code>FIND {MyProspect} RETURNING Contact(FirstName, LastName LIMIT 10), Account(Name, Industry)</code>
Multiple sObject objects, mixed number of fields	<code>FIND {MyProspect} RETURNING Contact(FirstName, LastName), Account, Lead(FirstName)</code>
Unsearchable sObject objects	<code>FIND {MyProspect} RETURNING RecordType</code>
	<code>FIND {MyProspect} RETURNING Pricebook</code>
Invalid sObject objects	<code>FIND {MyProspect} RETURNING FooBar</code>
Invalid sObject field	<code>FIND {MyProspect} RETURNING Contact(fooBar)</code>
Single object limit	<code>FIND {MyProspect} RETURNING Contact(FirstName, LastName LIMIT 10)</code>
Multiple object limits and a query limit	<code>FIND {MyProspect} RETURNING Contact(FirstName, LastName LIMIT 20), Account(Name, Industry LIMIT 10), Opportunity LIMIT 50</code>
Single object offset	<code>FIND {MyProspect} RETURNING Contact(FirstName, LastName OFFSET 10)</code>
List view	<code>FIND {MyAccount} IN ALL FIELDS RETURNING Account(Id, Name USING ListView=ListViewName)</code>



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

toLabel(fields)

Use `toLabel(fields)` to translate SOSL query results into the user's language.

Any organization can use the `toLabel()` method. It's particularly useful for organizations that enabled the Translation Workbench.

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

For example:

```
FIND {Joe} RETURNING Lead(company, toLabel(Recordtype.Name))
```

The returned records are translated into the language of 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 the `toLabel()` method to filter records using a translated picklist value. For example:

```
FIND {test} RETURNING Lead(company, toLabel(Status) WHERE toLabel(Status) = 'le Draft'
)
```

The query returns lead records with a picklist value for Status equal to 'le Draft.' The comparison is made against the value for the user's language. If no translation is available for the 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

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 *conditionExpression* 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. You must specify parentheses when nesting operators. However, multiple operators of the same type do not need to be nested. In the following example, 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
```

fieldExpression

A *fieldExpression* uses the following syntax:

```
fieldName comparisonOperator value
```



where:

Syntax	Description
<i>fieldName</i>	The name of a field for the specified object. Use of single or double quotes around the name 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. The name doesn't need to be a field in the <i>fieldList</i> .
<i>comparisonOperator</i>	Operators that compare values, such as =, <=, IN, and LIKE. Operators are case insensitive for most fields, but case sensitive for case-sensitive fields.
<i>value</i>	A value used to compare with the value in the <i>fieldName</i> . Supply a value whose data type matches the field type of the specified field. The value must be a valid value, not other field names or calculations. If quotes are required, use single quotes. Double quotes result in an error. Quotes are unnecessary for dates and numbers.

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 <i>fieldName</i> equals the <i>value</i> in the expression.
!=	Not equals	Expression is true if the value in the <i>fieldName</i> doesn't equal the specified <i>value</i> .
<	Less than	Expression is true if the value in the <i>fieldName</i> is less than the specified <i>value</i> .
<=	Less or equal	Expression is true if the value in the <i>fieldName</i> is less than or equal to the specified <i>value</i> .
>	Greater than	Expression is true if the value in the <i>fieldName</i> is greater than the specified <i>value</i> .
>=	Greater or equal	Expression is true if the value in the <i>fieldName</i> is greater than or equal to the specified <i>value</i> .
LIKE	Like	<p>Expression is true if the value in the <i>fieldName</i> matches the characters of the text string in the specified <i>value</i>. The text string in the specified value must be enclosed in single quotes.</p> <p>The LIKE operator is supported for string fields only. The operator provides a mechanism for matching partial text strings and includes support for:</p> <ul style="list-style-type: none"> The % and _ wildcards . <ul style="list-style-type: none"> The % wildcard matches zero or more characters. The _ wildcard matches exactly one character.

Operator	Name	Description
		<ul style="list-style-type: none"> The escaping of special characters % or _. <p> Note: Don't use the backslash (\) character in a search except to escape a special character. See Quoted String Escape Sequences.</p> <p>This example query matches Appleton, Apple, and Appl, but not Bappl.</p> <pre>SELECT AccountId, FirstName, lastname FROM Contact WHERE lastname LIKE 'appl%'</pre>
IN	IN	<p>If the value equals any one of the values in a <code>WHERE</code> clause. The string values for <code>IN</code> must be in parentheses and surrounded by single quotes.</p> <p>You can query values in a field where another field on the same object has a specified set of values, using <code>IN</code>. For example:</p> <pre>SELECT Name FROM Account WHERE BillingState IN ('California', 'New York')</pre> <p>You can also use <code>IN</code> and <code>NOT IN</code> for semi-joins and anti-joins when querying on ID (primary key) or reference (foreign key) fields.</p>
NOT IN	NOT IN	<p>If the value doesn't equal any of the values in a <code>WHERE</code> clause. The string values for <code>NOT IN</code> must be in parentheses and surrounded by single quotes. For example:</p> <pre>SELECT Name FROM Account WHERE BillingState NOT IN ('California', 'New York')</pre> <p> Note: The logical operator <code>NOT</code> is unrelated to this comparison operator.</p>
INCLUDES EXCLUDES		Applies only to multi-select picklists. See Query Multi-Select Picklists .

Logical Operators

The following table lists the logical operator values that are used in *fieldExpression* syntax:

Operator	Syntax	Description
AND	<i>fieldExpressionX</i> AND <i>fieldExpressionY</i>	true if both <i>fieldExpressionX</i> and <i>fieldExpressionY</i> are true.
OR	<i>fieldExpressionX</i> OR <i>fieldExpressionY</i>	<p>true if either <i>fieldExpressionX</i> or <i>fieldExpressionY</i> is true.</p> <p>In a <code>WHERE</code> clause that uses <code>OR</code>, records are returned even if the foreign key value in a record is null.</p> <pre>SELECT Id FROM Contact WHERE LastName = 'Young' or Account.Name = 'Quarry'</pre>

Operator	Syntax	Description
NOT	not <i>fieldExpressionX</i>	true if <i>fieldExpressionX</i> is false. There is also a comparison operator NOT 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 (%)
\uXXXX	A Unicode character with XXXX as the code (for example, \u00e9 represents the é 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)
```

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', 'ka0D000000002HClAY'))
```

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)
```

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
<i>groupName</i>	The name of the data category group to filter. For information on category groups, see “Create and Modify Category Groups” in the Salesforce Help.
<i>Operator</i>	Use one of the following operators: <ul style="list-style-type: none"> • 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.
<i>category</i>	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 *ObjectTypeName*:

- 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.	<pre>FIND {tourism} RETURNING KnowledgeArticleVersion (Id, Title WHERE PublishStatus='online') WITH DATA CATEGORY Location__c AT America__c</pre>
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
	<pre>WITH DATA CATEGORY Geography__c ABOVE France__c AND Product__c AT mobile_phones__c</pre>
Search archived FAQ articles from one category group.	<pre>FIND {tourism} RETURNING FAQ__kav (Id, Title WHERE PublishStatus='archived') WITH DATA CATEGORY Geography__c AT Iceland__c</pre>
Search all draft Salesforce Knowledge articles from one category group.	<pre>FIND {tourism} RETURNING KnowledgeArticleVersion (Id, Title WHERE PublishStatus='draft') WITH DATA CATEGORY Geography__c BELOW Europe__c</pre>

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/objects/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 *NetworkIdSpec*

You can search for Experience Cloud site users and feeds by using the `WITH NETWORK` optional clause on a SOSL query. When you're filtering search results by Experience Cloud site, each site is represented by a `NetworkId`.

You can use the following syntax.

- `WITH NETWORK IN ('NetworkId1', 'NetworkId2', ...)` supports filtering by one or more Experience Cloud sites.
- `WITH NETWORK = 'NetworkId'` supports filtering by a single Experience Cloud site only.

For objects other than users and feeds, search results include matches across all Experience Cloud sites and internal company data, even if you use network filtering in your query.

- You can run searches against multiple objects in the same Experience Cloud site.
- You can't run scoped and unscoped searches in the same query. For example, you can't search users from an Experience Cloud site along with accounts from the entire org.

To filter search results for *groups* or *topics* by Experience Cloud site, use the `WHERE` clause with a `NetworkId` value. If you want to search for an internal org, use an all zero value for `NetworkId`.

Example WITH NETWORK *NetworkIdSpec* Clauses

To search multiple Experience Cloud sites 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 NetworkId 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 org 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 = '000000000000000'
```

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 = '01sxx0000002MffAAE'
```

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*.

```
FIND {San Francisco} IN ALL FIELDS RETURNING KnowledgeArticleVersion(id, title WHERE
PublishStatus = 'Online' AND Language = 'en_US') WITH
    SNIPPET (target_length=120)
```

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/subjects/KnowledgeArticleVersion/kaKD00000000001MAA"
  },
  "Id" : "kaKD00000000001MAA",
  "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/subjects/KnowledgeArticleVersion/kaBD00000000007DMAQ"
  },
  "Id" : "kaBD00000000007DMAQ",
  "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/subjects/KnowledgeArticleVersion/ka3D00000000042OIAQ"
  },
  "Id" : "ka3D00000000042OIAQ",
  "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 ObjectName` clause, with a `WHERE` clause that filters on the `PublishStatus` field.

In the `RETURNING` clause, specify one of the following for *ObjectName*:

- To search a specific article type, use the article type name with the suffix `__kav`.
- 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 &lt;a
href='https://salesforce.com'&gt;salesforce.com&lt;/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.

```
FIND {San Francisco} IN ALL FIELDS RETURNING KnowledgeArticleVersion(id, title WHERE
PublishStatus = 'Online' AND Language = 'en_US') WITH
    SNIPPET(target_length=120)
```

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

Anti-join [20](#)
Apex and SOSL [101](#)

B

Big Objects
 Querying [83](#)

C

Characters reserved in SOSL FIND [100](#)

F

FIND and Apex [101](#)

R

Reserved characters in SOSL FIND [100](#)

S

SearchQuery character limit [100](#)
Semi-join [20](#)
SOSL
 Apex syntax [101](#)
SOSL character limit [100](#)