Version 30.0: Spring '14

Database.com SOQL and SOSL Reference

Last updated: June 2, 2014

Table of Contents

Chapter 1

Salesforce Object Query Language (SOQL)

In this chapter ...

Use the Salesforce Object Query Language (SOQL) to construct simple but powerful query strings in the following environments:

- In the `queryString` parameter in the `query()` call
- In Apex statements
- In the Schema Explorer of the Force.com IDE

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

Note: SOQL does not support all advanced features of the SQL SELECT command. For example, you cannot 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 may 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 Widget__c records if the value of `Name` is `Ring`:

```
SELECT Id, Name
FROM Widget__c
WHERE Name = 'Ring'
```

Note: Apex requires that you surround SOQL and SOSL statements with square brackets in order to use them on the fly. Additionally, Apex script variables and expressions can be used if preceded by a colon (`:`).

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

SOQL Typographical Conventions

Topics about SOQL use the following typographical conventions:

Convention	Description
`SELECT Name FROM Widget__c`	In an example, 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 question marks and square brackets.
`SELECT` ***`fieldname`*** `FROM` ***`objectname`***	In an example or syntax statement, italics represent variables. You supply the actual value.
`?`	In a syntax statement, the question mark indicates the element preceding it is optional. You may omit the element or include one.
`WHERE [`***`conditionexpression`***`]`	In a syntax statement, square brackets surround an element that may be repeated up to the limits for that element. You may omit the element, or include one or more of them.
`SELECT Name FROM` **`Widget__c`**	In some examples, particular elements are highlighted with bold if they are of particular interest in the text before or after the example.

Alias Notation

You can use alias notation in SELECT queries:

```
SELECT count()
FROM Model__c m, m.Widget__r w
WHERE w.Name = 'Ring'
```

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

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

Quoted String Escape Sequences

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

Sequence	Meaning
`\'`	One single-quote character
`\\`	Backslash
LIKE expression only: `\_`	Matches a single underscore character (_)
LIKE expression only:`\%`	Matches a single percent sign character (%)

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

Escaped Character Examples

`SELECT Id FROM Widget__C WHERE Name LIKE R%'`

Select all widgets whose name begins with the letter 'R'.

`SELECT Id FROM Widget__c WHERE Name LIKE 'R\%'`

Select all widgets whose name exactly matches the two character sequence 'R%'.

`SELECT Id FROM Widget__c WHERE Name LIKE 'R\%%'`

Select all widgets whose name begins with the two character sequence 'R%'

Reserved Characters

Reserved characters, if specified in a `SELECT` clause as a literal string (between single quotes), must be escaped (preceded by the backslash \ character) in order 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 Widget__c `Name` field for “Wally’s grommet,” use the following SELECT statement:

```
SELECT Id
FROM Widget__c
WHERE Name LIKE 'Wally\'s grommet'
```

SOQL SELECT Syntax

The SOQL SELECT statement uses the following syntax:

```
SELECT fieldList
[TYPEOF typeOfField whenExpression elseExpression END]
FROM objectType
[WHERE conditionExpression]
[GROUP BY fieldGroupByList]]
[HAVING havingConditionExpression]
[ORDER BY fieldOrderByList ASC | DESC ? NULLS FIRST | LAST ?]
[LIMIT ?]
[OFFSET ?]
[UPDATE VIEWSTAT ?]
```

Note: `TYPEOF` is currently available as a Developer Preview as part of the SOQL Polymorphism feature. For more information on enabling `TYPEOF` for your organization, contact salesforce.com.

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`*s:
	• `SELECT` **`Id, Name, Widget_Cost__c`** `FROM Widget__c`
	• `SELECT` **`count()`** `FROM Model__c`
	• `SELECT` **`Model__c.Name, Model__c.Widget__r.Name`** `FROM Model__c`
	You must specify valid field names and must have read-level permissions to each specified field. The *`fieldList`* defines the ordering of fields in the query results.
	`fieldList` can include a subquery if the query traverses a relationship. For example:
	`SELECT Widget__c.Name, (SELECT Model__c.Name FROM`
	`Widget__c.Models__r)`
	`FROM Widget__c`
	The *`fieldlist`* can also be an aggregate function, such as `COUNT()` and `COUNT(`***`fieldName`***`)`.

`typeOfField`	A polymorphic relationship field in ***`objectType`*** or a polymorphic field in a parent of ***`objectType`*** that can reference multiple object types. For example, the Owner relationship field of a Merchandise__c custom object could be a User or a Group. ***`typeOfField`*** cannot reference a relationship field that is also referenced in ***`fieldList`***. See TYPEOF for more information.
`whenExpression`	A clause of the form `WHEN` ***`whenObjectType`*** `THEN` ***`whenFieldList`***. You can have one or more ***`whenExpression`*** clauses inside a `TYPEOF` expression. See TYPEOF for more information.
`elseExpression`	A clause of the form `ELSE` ***`elseFieldList`***. This clause is optional inside a `TYPEOF` expression. See TYPEOF for more information.
`objectType`	Specifies the type of object that you want to `query()`. You must specify a valid object, such as Widget__c, and must have read-level permissions to that object.
`conditionExpression`	If `WHERE` is specified, determines which rows and values in the specified object (*`objectType`*) to filter against. If unspecified, the `query()` retrieves all the rows in the object that are visible to the user.
`fieldGroupByList`	Available in API version 18.0 and later. Specifies a list of one or more fields, separated by commas, that are used to group the query results. A GROUP BY clause is used with aggregate functions to summarize the data and enable you to roll up query results rather than having to process the individual records in your code. See GROUP BY.
`havingConditionExpression`	Available in API version 18.0 and later. If the query includes a `GROUP BY` clause, this conditional expression filters the records that the GROUP BY returns. See HAVING.

Syntax	Description

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

Note the following implementation tips:

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

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

- **Localized Results**—SELECT statements can include the `convertCurrency()` functions in support of localized fields.
- **Dynamic SOQL in Apex**—Apex requires that you surround SOQL and SOSL statements with square brackets in order to use them on the fly. Additionally, Apex script variables and expressions can be used if preceded by a colon (`:`).
- **Ordered Results**—There is no guarantee of the order of results unless you use an ORDER BY clause in a query.

Condition Expression Syntax (`WHERE` Clause)

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

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

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

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

- `SELECT Name FROM Widget__c WHERE Name like 'R%'`
- `SELECT Id FROM Model__c` **`WHERE Name like 'G%' AND Model_Number__c = 'XL-5'`**
- `SELECT Name FROM Widget__c` **`WHERE CreatedDate > 2011-04-26T10:00:00-08:00`**

  You can use date or datetime values, or date literals. The format for date and dateTime fields are different.
- `SELECT Widget_Cost__c FROM Widget__c` **`WHERE CALENDAR_YEAR(CreatedDate) = 2011`**

  For more information on date functions, such as `CALENDAR_YEAR()`, see Date Functions.
- You can use parentheses to define the order in which *`fieldExpression`*s are evaluated. For example, the following expression is `true` if `fieldExpression1` is `true` and either `fieldExpression2` or `fieldExpression3` are `true`:

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

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

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

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

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

```
SELECT Id
FROM Model__c
WHERE Widget__r.name = null
```

`null` in SOQL Queries

Use the value `null` to represent null values in SOQL queries.

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

```
SELECT Id
FROM Widget__c
WHERE Name != null
```

Filtering on Boolean Fields

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

```
WHERE BooleanField = TRUE

WHERE BooleanField = FALSE
```

Querying Multi-Select Picklists

Client applications use a specific syntax for querying multi-select picklists (in which multiple items can be selected).

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

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

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

Examples

The following query filters on values in the `MSP1__c` field that are equal to `AAA` and `BBB` selected (exact match):

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

In the following query:

```
SELECT Id, MSP1__c from CustObj__c WHERE MSP1__c includes ('AAA;BBB','CCC')
```

the query filters on values in the `MSP1__c` field that contains either of these values:

- `AAA` and `BBB` selected.
- `CCC` selected.

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

- matches with ' `AAA;BBB`':

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

- matches with ' `CCC`':

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

Filtering on Polymorphic Relationship Fields

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

```
WHERE polymorphicRelationship.Type comparisonExpression
```

Syntax	Description
polymorphicRelationship	A polymorphic relationship field in object being queried that can reference multiple object types. For example, the Owner relationship field of a Merchandise__c custom object could be a User or a Group.
comparisonExpression	The comparison being made against the object type in the polymorphic relationship. For more information, see *`fieldExpression`* Syntax. Note that the type names returned by Type are string values, like 'User'.

The following example only returns records where the Owner field of Merchandise__c is referencing a User or Group.

```
SELECT Description
FROM Merchandise__c
WHERE Owner.Type IN ('User', 'Group')
```

See Understanding Polymorphic Keys and Relationships for more details on polymorphic relationships, and additional filtering examples.

`fieldExpression` Syntax

`fieldExpression` uses the following syntax:

```
fieldName comparisonOperator value
```

where:

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

Comparison Operators

The following table lists the *`comparisonOperator`* values that are used in *`fieldExpression`* syntax. Note that comparisons on strings are case-insensitive.

Operator	Name	Description
`=`	`Equals`	Expression is true if the value in the specified *`fieldName`* equals the specified *`value`* in the expression. String comparisons using the equals operator are case-insensitive.
`!=`	`Not equals`	Expression is true if the value in the specified *`fieldName`* does not equal the specified *`value`*.
`<`	`Less than`	Expression is true if the value in the specified *`fieldName`* is less than the specified *`value`*.
`<=`	`Less or equal`	Expression is true if the value in the specified *`fieldName`* is less than, or equals, the specified *`value`*.
`>`	`Greater than`	Expression is true if the value in the specified *`fieldName`* is greater than the specified *`value`*.

Operator	Name	Description
`>=`	`Greater or equal`	Expression is true if the value in the specified *`fieldName`* is greater than or equal to the specified *`value`*.

`LIKE`	`Like`	Expression is true if the value in the specified *`fieldName`* matches the characters of the text string in the specified *`value`*. The `LIKE` operator in SOQL and SOSL is similar to the `LIKE` operator in SQL; it provides a mechanism for matching partial text strings and includes support for wildcards.
		• The `%` and _ wildcards are supported for the `LIKE` operator.
		• The `%` wildcard matches zero or more characters.
		• The _ wildcard matches exactly one character.
		• The text string in the specified *`value`* must be enclosed in single quotes.
		• The `LIKE` operator is supported for string fields only.
		• The `LIKE` operator performs a case-insensitive match, unlike the case-sensitive matching in SQL.
		• The `LIKE` operator in SOQL and SOSL supports escaping of special characters `%` or _.
		• Do not use the backslash character in a search except to escape a special character.
		For example, the following query matches Appleton, Apple, and Appl, but not Bappl:
		`SELECT Id, Name`
		`FROM Widget__c`
		`WHERE Name LIKE 'appl%'`

`IN`	`IN`	If the value equals any one of the specified values in a `WHERE` clause. For example:
		`SELECT Name FROM Widget__c`
		`WHERE Name IN ('Ring', 'Flange')`
		Note that the values for `IN` must be in parentheses. String values must be surrounded by single quotes.
		`IN` and `NOT IN` can also be used for semi-joins and anti-joins when querying on ID (primary key) or reference (foreign key) fields.

`NOT IN`	`NOT IN`	If the value does not equal any of the specified values in a `WHERE` clause. For example:
		`SELECT Name FROM Widget__c`
		`WHERE Name NOT IN ('Ring', 'Flange')`
		Note that the values for `NOT IN` must be in parentheses, and string values must be surrounded by single quotes.
		There is also a logical operator `NOT`, which is unrelated to this comparison operator.

`INCLUDES` `EXCLUDES`		Applies only to multi-select picklists.

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

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

```
SELECT Name FROM Widget__c
WHERE Name IN ('Ring', 'Flange')
```

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

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

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 User
WHERE Id IN
(
  SELECT OwnerId
  FROM Widget__c
  WHERE Name = 'Ring'
)
```

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

Reference Field Semi-Join

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

```
SELECT Id, Name
FROM Widget__c
WHERE OwnerId IN
  (
    SELECT Id
    FROM User
    WHERE FirstName= 'Sydney'
)
```

Notice that the left operand, `OwnerId`, of the `IN` clause is a reference field. An interesting aspect of this query is that `OwnerId` 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 user IDs for all users who do not have any widgets called'Ring':

```
SELECT Id
FROM User
WHERE Id NOT IN
(
  SELECT OwnerId
  FROM Widget__c
  WHERE Name = 'Ring'
)
```

Reference Field Anti-Join

The following query returns user IDs for all widgets whose first name is not 'Sydney':

```
SELECT Id
FROM Widget__c
WHERE OwnerId NOT IN
  (
    SELECT Id
    FROM User
    WHERE FirstName= 'Sydney'
)
```

This is a child-to-parent anti-join from `Widget` to `User`.

Multiple Semi-Joins or Anti-Joins

You can combine semi-join or anti-join clauses in a query. For example, the following query returns Widget IDs that have Parts with Part Numbers that begin with 'X' and Models with Names that begin with 'G'.

```
SELECT Id, Name
FROM Widget__c
WHERE Id IN
  (
    SELECT Widget__c
    FROM Part__c
    WHERE Part_Number__c LIKE 'X%')
    AND Id IN
  (
    SELECT Widget__c
    FROM Model__c
    WHERE Name LIKE 'G%'
  )
```

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 Widget IDs and their related Models if the Widget's Model has a Name that begins with 'G':

```
SELECT Id, (SELECT Id from Models__r)
FROM Widget__c
WHERE Id IN
  (
    SELECT Widget__c
    FROM Model__c
    WHERE Name LIKE 'G%'
  )
```

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

- **Basic limits:**
  - ◊ No more than two `IN` or `NOT IN` statements per `WHERE` clause.
  - ◊ You cannot use the `NOT` operator as a conjunction with semi-joins and anti-joins. Using them converts a semi-join to an anti-join, and vice versa. 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 Model__c
WHERE Widget__c.Id IN
  (
   SELECT ...
  )
```

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

```
SELECT Id, Name
FROM Widget__c
WHERE Id IN
  (
    SELECT Widget__c
    FROM Model__c
    WHERE Name LIKE 'G%'
  )
```

Using `Owner.Id` (dot notation) instead of OwnerId is not supported.

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

```
SELECT Id, Name
FROM Model__c
WHERE Id IN
  (
    SELECT Widget__c.Id
    FROM Widget__c
    WHERE Name = 'Ring'
  )
```

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

```
SELECT Id, Name
FROM Model__c
WHERE Widget__r.Name = 'Ring'
```

  - ◊ You cannot nest a semi-join or anti-join statement in another semi-join or anti-join statement.

◊ You can use semi-joins and anti-joins in the main `WHERE` statement, but not in a subquery `WHERE` statement. For example, the following query is valid:

```
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'))
```

```
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 additional 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 cannot use subqueries in conjunction with `OR`.

◊ `COUNT`, `FOR UPDATE`, `ORDER BY`, and `LIMIT` are not supported in subqueries.

◊ The following objects are not currently supported in subqueries:

- ActivityHistory
- Attachments
- Event
- EventAttendee
- Note
- OpenActivity
- Tags (AccountTag, ContactTag, and all other tag objects)
- Task

Logical Operators

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

Operator	Syntax	Description
`AND`	***`fieldExpressionX`*** `AND` ***`fieldExpressionY`***	`true` if both *`fieldExpressionX`* and *`fieldExpressionY`* are true.

Operator	Syntax	Description
`OR`	***`fieldExpressionX`*** `OR` ***`fieldExpressionY`***	`true` if either *`fieldExpressionX`* or *`fieldExpressionY`* is true. Relationship queries with foreign key values in an `OR` clause behave differently depending on the version of the API. In a WHERE clause using `OR`, if the foreign key value in a record is null, the record is returned in version 13.0 and later, but not returned in versions before 13.0. `SELECT Id FROM Model__c WHERE Name = 'Wallace' OR Widget__r.Name = 'Ring'` The Model with no parent Widget has a name that meets the criteria, so it is returned in version 13.0 and later.
`NOT`	`not` ***`fieldExpressionX`***	`true` if *`fieldExpressionX`* is false. There is also a comparison operator `NOT IN`, which is different from this logical operator.

Date Formats and Date Literals

When you specify a date in a SOQL query, it can be a specific date, or a date literal, which is a fixed expression representing a relative range of time such as last month or next year. Remember that dateTime field values are stored as Coordinated Universal Time (UTC). When one of these values is returned in Database.com, it is automatically adjusted for the time zone specified in your organization preferences. SOQL queries return dateTime field values as UTC values. Your application may need to handle the conversion if you want to process these values in different time zones.

Date Formats

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

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

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

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

Note: For a *`fieldExpression`* that uses date formats, the date is not enclosed in single quotes. No quotes should be used around the date. For example:

```
SELECT Id
FROM Widget__c
WHERE CreatedDate > 2010-10-08T01:02:03Z
```

Date Literals

A *`fieldExpression`* can use a date literal to compare a range of values to the value in a `date` or `dateTime` field. Each literal is a range of time beginning with midnight (12:00:00). To find a value within the range, use =. To find values on either side of the range, use > or <. The following table shows the available list of date literals, the ranges they represent, and examples:

Date Literal	Range	Example
`YESTERDAY`	Starts 12:00:00 the day before and continues for 24 hours.	`SELECT Id FROM Widget__c WHERE CreatedDate = YESTERDAY`
`TODAY`	Starts 12:00:00 of the current day and continues for 24 hours.	`SELECT Id FROM Widget__c WHERE CreatedDate > TODAY`
`TOMORROW`	Starts 12:00:00 after the current day and continues for 24 hours.	`SELECT Id FROM Model__c WHERE LastModifiedDate = TOMORROW`
`LAST_WEEK`	Starts 12:00:00 on the first day of the week before the most recent first day of the week and continues for seven full days. First day of the week is determined by your locale.	`SELECT Id FROM Widget__c WHERE CreatedDate > LAST_WEEK`
`THIS_WEEK`	Starts 12:00:00 on the most recent first day of the week before the current day and continues for seven full days. First day of the week is determined by your locale.	`SELECT Id FROM Widget__c WHERE CreatedDate < THIS_WEEK`
`NEXT_WEEK`	Starts 12:00:00 on the most recent first day of the week after the current day and continues for seven full days. First day of the week is determined by your locale.	`SELECT Id FROM Model__c WHERE LastModifiedDate = NEXT_WEEK`
`LAST_MONTH`	Starts 12:00:00 on the first day of the month before the current day and continues for all the days of that month.	`SELECT Id FROM Model__c WHERE LastModifiedDate > LAST_MONTH`
`THIS_MONTH`	Starts 12:00:00 on the first day of the month that the current day is in and continues for all the days of that month.	`SELECT Id FROM Widget__c WHERE CreatedDate < THIS_MONTH`
`NEXT_MONTH`	Starts 12:00:00 on the first day of the month after the month that the current day is in and continues for all the days of that month.	`SELECT Id FROM Model__c WHERE LastModifiedDate = NEXT_MONTH`
`LAST_90_DAYS`	Starts 12:00:00 of the current day and continues for the last 90 days.	`SELECT Id FROM Widget__c WHERE LastModifiedDate = LAST_90_DAYS`
`NEXT_90_DAYS`	Starts 12:00:00 of the current day and continues for the next 90 days.	`SELECT Id FROM Model__c WHERE LastModifiedDate > NEXT_90_DAYS`
`LAST_N_DAYS:`***`n`***	For the number *n* provided, starts 12:00:00 of the current day and continues for the last *n* days.	`SELECT Id FROM Widget__c WHERE CreatedDate = LAST_N_DAYS:365`
`NEXT_N_DAYS:`***`n`***	For the number *n* provided, starts 12:00:00 of the current day and continues for the next *n* days.	`SELECT Id FROM Model__c WHERE LastModifiedDate > NEXT_N_DAYS:15`
`NEXT_N_WEEKS:`***`n`***	For the number *n* provided, starts 12:00:00 of the first day of the next week and continues for the next *n* weeks.	`SELECT Id FROM Merchandise__c WHERE LastModifiedDate > NEXT_N_WEEKS:4`

Date Literal	Range	Example
`LAST_N_WEEKS:`***n***	For the number *n* provided, starts 12:00:00 of the last day of the previous week and continues for the last *n* weeks.	`SELECT Id FROM Merchandise__c WHERE CreatedDate = LAST_N_WEEKS:52`
`NEXT_N_MONTHS:`***n***	For the number *n* provided, starts 12:00:00 of the first day of the next month and continues for the next *n* months.	`SELECT Id FROM Merchandise__c WHERE LastModifiedDate > NEXT_N_MONTHS:2`
`LAST_N_MONTHS:`***n***	For the number *n* provided, starts 12:00:00 of the last day of the previous month and continues for the last *n* months.	`SELECT Id FROM Merchandise__c WHERE CreatedDate = LAST_N_MONTHS:12`
`THIS_QUARTER`	Starts 12:00:00 of the current quarter and continues to the end of the current quarter.	`SELECT Id FROM Widget__c WHERE CreatedDate = THIS_QUARTER`
`LAST_QUARTER`	Starts 12:00:00 of the previous quarter and continues to the end of that quarter.	`SELECT Id FROM Widget__c WHERE CreatedDate > LAST_QUARTER`
`NEXT_QUARTER`	Starts 12:00:00 of the next quarter and continues to the end of that quarter.	`SELECT Id FROM Widget__c WHERE CreatedDate < NEXT_QUARTER`
`NEXT_N_QUARTERS:`***n***	Starts 12:00:00 of the next quarter and continues to the end of the *n*th quarter.	`SELECT Id FROM Widget__c WHERE CreatedDate < NEXT_N_QUARTERS:2`
`LAST_N_QUARTERS:`***n***	Starts 12:00:00 of the previous quarter and continues to the end of the previous *n*th quarter.	`SELECT Id FROM Widget__c WHERE CreatedDate > LAST_N_QUARTERS:2`
`THIS_YEAR`	Starts 12:00:00 on January 1 of the current year and continues through the end of December 31 of the current year.	`SELECT Id FROM Model__c WHERE LastModifiedDate = THIS_YEAR`
`LAST_YEAR`	Starts 12:00:00 on January 1 of the previous year and continues through the end of December 31 of that year.	`SELECT Id FROM Model__c WHERE LastModifiedDate > LAST_YEAR`
`NEXT_YEAR`	Starts 12:00:00 on January 1 of the following year and continues through the end of December 31 of that year.	`SELECT Id FROM Model__c WHERE LastModifiedDate < NEXT_YEAR`
`NEXT_N_YEARS:`***n***	Starts 12:00:00 on January 1 of the following year and continues through the end of December 31 of the *n*th year.	`SELECT Id FROM Model__c WHERE LastModifiedDate < NEXT_N_YEARS:5`
`LAST_N_YEARS:`***n***	Starts 12:00:00 on January 1 of the previous year and continues through the end of December 31 of the previous *n*th year.	`SELECT Id FROM Model__c WHERE LastModifiedDate > LAST_N_YEARS:5`
`THIS_FISCAL_QUARTER`	Starts 12:00:00 on the first day of the current fiscal quarter and continues through the end of the last day of the fiscal quarter. The fiscal year is defined in the company profile under Setup at **Company Profile** > **Fiscal Year**.	`SELECT Id FROM Widget__c WHERE CreatedDate = THIS_FISCAL_QUARTER`
`LAST_FISCAL_QUARTER`	Starts 12:00:00 on the first day of the last fiscal quarter and continues through the end of the last day of that fiscal quarter. The fiscal year is defined in the company profile under Setup at **Company Profile** > **Fiscal Year**.	`SELECT Id FROM Widget__c WHERE CreatedDate > LAST_FISCAL_QUARTER`

Date Literal	Range	Example
`NEXT_FISCAL_QUARTER`	Starts 12:00:00 on the first day of the next fiscal quarter and continues through the end of the last day of that fiscal quarter. The fiscal year is defined in the company profile under Setup at **Company Profile** > **Fiscal Year**.	`SELECT Id FROM Widget__c WHERE CreatedDate < NEXT_FISCAL_QUARTER`
`NEXT_N_FISCAL_QUARTERS:`***n***	Starts 12:00:00 on the first day of the next fiscal quarter and continues through the end of the last day of the *n*th fiscal quarter. The fiscal year is defined in the company profile under Setup at**Company Profile** > **Fiscal Year**.	`SELECT Id FROM Widget__c WHERE CreatedDate < NEXT_N_FISCAL_QUARTERS:6`
`LAST_N_FISCAL_QUARTERS:`***n***	Starts 12:00:00 on the first day of the last fiscal quarter and continues through the end of the last day of the previous *n*th fiscal quarter. The fiscal year is defined in the company profile under Setup at **Company Profile** > **Fiscal Year**.	`SELECT Id FROM Widget__c WHERE CreatedDate > LAST_N_FISCAL_QUARTERS:6`
`THIS_FISCAL_YEAR`	Starts 12:00:00 on the first day of the current fiscal year and continues through the end of the last day of the fiscal year. The fiscal year is defined in the company profile under Setup at **Company Profile** > **Fiscal Year**.	`SELECT Id FROM Model__c WHERE LastModifiedDate = THIS_FISCAL_YEAR`
`LAST_FISCAL_YEAR`	Starts 12:00:00 on the first day of the last fiscal year and continues through the end of the last day of that fiscal year. The fiscal year is defined in the company profile under Setup at **Company Profile** > **Fiscal Year**.	`SELECT Id FROM Model__c WHERE LastModifiedDate > LAST_FISCAL_YEAR`
`NEXT_FISCAL_YEAR`	Starts 12:00:00 on the first day of the next fiscal year and continues through the end of the last day of that fiscal year. The fiscal year is defined in the company profile under Setup at **Company Profile** > **Fiscal Year**.	`SELECT Id FROM Model__c WHERE LastModifiedDate < NEXT_FISCAL_YEAR`
`NEXT_N_FISCAL_YEARS:`***n***	Starts 12:00:00 on the first day of the next fiscal year and continues through the end of the last day of the *n*th fiscal year. The fiscal year is defined in the company profile under Setup at **Company Profile** > **Fiscal Year**.	`SELECT Id FROM Model__c WHERE LastModifiedDate < NEXT_N_FISCAL_YEARS:3`
`LAST_N_FISCAL_YEARS:`***n***	Starts 12:00:00 on the first day of the last fiscal year and continues through the end of the last day of the previous *n*th fiscal year. The fiscal year is defined in the company profile under Setup at **Company Profile** > **Fiscal Year**.	`SELECT Id FROM Model__c WHERE LastModifiedDate > LAST_N_FISCAL_YEARS:3`

Note: If you have defined **Custom Fiscal Years** in the Database.com user interface, and in any of the `FISCAL` date literals you specify a range that is outside the years you've defined, an invalid date error is returned.

Minimum and Maximum Dates

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

ORDER BY

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

```
[ORDER BY fieldExpression 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.

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

```
SELECT Name
FROM Widget__c
ORDER BY Name DESC NULLS LAST
```

The following factors affect results returned with `ORDER BY`:

- Sorting is case insensitive.
- `ORDER BY` is compatible with relationship query syntax.
- Multiple column sorting is supported, by listing more than one *`fieldExpression`* clause.
- Relationship queries with foreign key values in an `ORDER BY` clause behave differently depending on the version of the Force.com API. In an `ORDER BY` clause, if the foreign key value in a record is null, the record is returned in version 13.0 and later, but not returned in versions before 13.0.

  ```
  SELECT Id, Name, Widget__r.Id, Widget__r.Name
  FROM Model__c
  ORDER BY Widget__r.Name
  ```

  Any Model record for which `Widget__c` is empty is returned in version 13.0 and later.
- Sort order is determined by current user locale. For English locales, Database.com uses a sorting mechanism based on the UTF-8 values of the character data. For Asian locales, Database.com uses a linguistic sorting mechanism based on the ISO 14651 and Unicode 3.2 standards.

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

- These data types are not supported: multi-select picklist, rich text area, long text area, and encrypted (if enabled).
- All other data types are supported, with the following caveats:
  - ◊ `convertCurrency()` 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.

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

```
SELECT Name
FROM Model__c
WHERE Name= 'Ring'
ORDER BY Model_Number__c ASC NULLS LAST LIMIT 125
```

You are limited to 32 fields in an `ORDER BY` query. If you exceed the limit, a malformed query fault is returned.

LIMIT

Use `LIMIT` to specify the maximum number of rows to return:

```
SELECT fieldList
FROM objectType
[WHERE conditionExpression]
  LIMIT number_of_rows
```

For example:

```
SELECT Name
FROM Widget__c
WHERE Name = 'Ring' LIMIT 125
```

This query returns the first 125 Widget records whose Name is `Ring`.

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 Wdiget__c LIMIT 1
```

OFFSET

Use `OFFSET` to specify the starting row offset into the result set returned by your query. Using `OFFSET` is helpful for paging into large result sets, in scenarios where you need to quickly jump to a particular subset of the entire results. As 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 number_of_rows_to_return
OFFSET number_of_rows_to_skip
```

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

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

The result set for the preceding example would be a subset of the full result set, returning rows 11 through 50 of the full set.

Considerations When Using `OFFSET`

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

- The maximum offset is 2,000 rows. Requesting an offset greater than 2,000 will result in a `NUMBER_OUTSIDE_VALID_RANGE` error.
- `OFFSET` is intended to be used in a top-level query, and is not allowed in most sub-queries, so the following query is invalid and will return 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 sub-query can use `OFFSET` only if the parent query has a `LIMIT 1` clause. The following query is a valid use of `OFFSET` in a sub-query:

  ```
  SELECT Name, Id
      (
          SELECT Name FROM Line_Items__r LIMIT 10 OFFSET 2
      )
  FROM Merchandise__c
  ORDER BY Name
  LIMIT 1
  ```

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

  Note: Using `OFFSET` in sub-queries 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.com, inc.

- 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 will have a stable ordering, however the ordering key is subject to change and should not be relied on.
- Similarly, 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:

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

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

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

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

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

  After the query is run, 10 new rows are then inserted into Merchandise__c with Name values that come early in the sort order. If the query is run again, with the same `OFFSET` value, a different set of rows is skipped. If you need to query multiple pages of records with a consistent server-side cursor, use the `queryMore()` in SOAP API.
- Offsets are not intended to be used as a replacement for using `queryMore()`, given the maximum offset size and lack of a server-side cursor. Multiple queries with offsets into a large result set will have a higher performance impact than using `queryMore()` against a server-side cursor.
- 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.
- The `OFFSET` clause is allowed in SOQL used in SOAP API, REST API, and Apex. It's not allowed in SOQL used within Bulk API or Streaming API.

UPDATE VIEWSTAT

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

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

```
SELECT Title FROM FAQ__kav
WHERE PublishStatus='online' and
Language = 'en_US' and
KnowledgeArticleVersion = 'ka230000000PCiy'
UPDATE VIEWSTAT
```

GROUP BY

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

```
[GROUP BY fieldGroupByList]
```

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

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

```
SELECT Widget__c FROM Model__c
```

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

```
SELECT Widget__c, COUNT(Name)
FROM Model__c
GROUP BY Widget__c
```

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

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

```
SELECT Widget__c
FROM Model__c
GROUP BY Widget__c
```

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

Considerations When Using `GROUP BY`

Note the following when you are creating queries with a `GROUP BY` clause:

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

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

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

  ```
  SELECT MAX(CreatedDate)
  FROM Wdiget__c 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.

`GROUP BY` and `queryMore()`

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

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

Using Aliases with `GROUP BY`

You can use an alias for any field or aggregated field in a `SELECT` list in a query. 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(Widget_Cost__c)` aggregated field.

```
SELECT Name n, MAX(Widget_Cost__c) max
FROM Widget__c
GROUP BY Name
```

You can use a field alias to identify the field when you are processing the query results in your code.

Any aggregated field in a `SELECT` list that does not have an alias automatically gets an implied alias with a format `expr`***`i`***, 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 the following example, `MAX(Widget_Cost__c)` has an implied alias of `expr0`, and `MIN(Widget_Cost__c)` has an implied alias of `expr1`.

```
SELECT Name, MAX(Widget_Cost__c), MIN(Widget_Cost__c)
FROM Widget__c
GROUP BY Name
```

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

```
SELECT Name, MAX(Widget_Cost__c), MIN(Widget_Cost__c) min, SUM(Widget_Cost__c)
FROM Widget__c
GROUP BY Name
```

HAVING

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

```
[HAVING havingConditionExpression]
```

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

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

```
SELECT Widget__c, COUNT(Name)
FROM Model__c
GROUP BY Widget__c
```

However, if you are only interested in Widgets that have more than 10 Models, you can filter the results by using a `HAVING` clause. For example:

```
SELECT Widget__c, COUNT(Name)
FROM Model__c
GROUP BY Widget__c
HAVING COUNT(Name) > 10
```

The next query returns Widgets with duplicate names:

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

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

Considerations When Using HAVING

Note the following when you are creating queries with a HAVING clause:

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

  ```
  SELECT Widget__c, COUNT(Name)
  FROM Model__c
  GROUP BY Widget
  HAVING COUNT(Name) > 10
  ```

  The following query is invalid as Model_Number__c is not included in the GROUP BY clause:

  ```
  SELECT Widget__c, COUNT(Name)
  FROM Model__c
  GROUP BY Widget__c
  HAVING COUNT(Name) > 10 and Model_Number__c LIKE 'Sirius%'
  ```

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

TYPEOF

Note: TYPEOF is currently available as a Developer Preview as part of the SOQL Polymorphism feature. For more information on enabling TYPEOF for your organization, contact salesforce.com.

Use TYPEOF in SELECT clauses when 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 26.0 and later.

```
SELECT [fieldList,]
    TYPEOF typeOfField
        WHEN whenObjectType THEN whenFieldList
        [ELSE elseFieldList]
    END
FROM objectType
```

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

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

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

Note the following considerations for using `TYPEOF`:

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

```
SELECT Name From Line_Item__c
WHERE CreatedById IN
    (
    SELECT
        TYPEOF Owner
            WHEN User THEN Id
            WHEN Group THEN CreatedById
        END
    FROM Merchandise__c
    )
```

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

```
SELECT
    TYPEOF Owner
        WHEN User THEN FirstName
        WHEN Group THEN Name
    END
FROM Line_Item__c
WHERE CreatedById IN
    (
    SELECT CreatedById
    FROM Merchandise__c
    )
```

- `GROUP BY` and `HAVING` aren't allowed in queries that use `TYPEOF`.

The following example selects specific fields depending on whether the Owner field of a Merchandise__c record references a User or Group.

```
SELECT
  TYPEOF Owner
    WHEN User THEN FirstName, LastName
    WHEN Group THEN Name, Email
    ELSE Name
  END
FROM Merchandise__c
```

See Understanding Polymorphic Keys and Relationships for more details on polymorphic relationships, and additional examples of `TYPEOF`.

Aggregate Functions

Aggregate functions allow you to roll up and summarize your data for analysis. You can use these 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(Widget_Cost__c)
FROM Widget__c
```

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 Id, AVG(Widget_Cost__c)
FROM Widget__c
GROUP BY Id
```

This table lists all the aggregate functions supported by SOQL.

Aggregate Function	Description

`AVG()`	Returns the average value of a numeric field. For example:
	`SELECT Id, AVG(Widget_Cost__c)`
	`FROM Widget__c`
	`GROUP BY Id`
	Available in API version 18.0 and later.

Aggregate Function	Description
`COUNT()` and `COUNT(`***`fieldName`***`)`	Returns the number of rows matching the query criteria. For example using `COUNT()`: `SELECT COUNT()` `FROM Widget__c` `WHERE Name LIKE 'Ring%'` For example using `COUNT(`***`fieldName`***`)`: `SELECT COUNT(Id)` `FROM Widget__c` `WHERE Name LIKE 'Ring%'` **Note:** `COUNT(Id)` in SOQL is equivalent to `COUNT(*)` in SQL. The `COUNT(`***`fieldName`***`)` syntax is available in API version 18.0 and later. If you are using a GROUP BY clause, use `COUNT(`***`fieldName`***`)` instead of `COUNT()`. For more information, see `COUNT()` and `COUNT(`***`fieldName`***`)`.
`COUNT_DISTINCT()`	Returns the number of distinct non-`null` field values matching the query criteria. For example: `SELECT COUNT_DISTINCT(Name)` `FROM Widget__c` **Note:** `COUNT_DISTINCT(`***`fieldName`***`)` in SOQL is equivalent to `COUNT(DISTINCT `***`fieldName`***`)` in SQL. To query for all the distinct values, including `null`, for an object, see GROUP BY. Available in API version 18.0 and later.
`MIN()`	Returns the minimum value of a field. For example: `SELECT MIN(CreatedDate), Name` `FROM Widget__c` `GROUP BY Name` If you use the `MIN()` or `MAX()` functions on a picklist field, the function uses the sort order of the picklist values instead of alphabetical order. Available in API version 18.0 and later.
`MAX()`	Returns the maximum value of a field. For example: `SELECT Name, MAX(Widget_Cost__c)` `FROM Widget__c` `GROUP BY Name` Available in API version 18.0 and later.
`SUM()`	Returns the total sum of a numeric field. For example: `SELECT SUM(Widget_Cost__c)` `FROM Widget__c`

Aggregate Function	Description
	Available in API version 18.0 and later.

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

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

`COUNT()` and `COUNT(`***`fieldName`***`)`

To discover the number of rows that are returned by a query, use `COUNT()` in a `SELECT` clause. There are two versions of syntax for `COUNT()`:

- `COUNT()`
- `COUNT(`***`fieldName`***`)`

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

COUNT()

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

For example:

```
SELECT COUNT()
FROM Widget__c
WHERE Name LIKE 'Ring%'
```

```
SELECT COUNT()
FROM Model__c, Model__c.Widget__r
WHERE Widget__r.Name = 'Ring'
```

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

Note the following when using `COUNT()`:

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

COUNT(*fieldName*)

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

For example:

```
SELECT COUNT(Id)
FROM Widget__c
WHERE Name LIKE 'Ring%'
```

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

```
SELECT COUNT()
FROM Widget__c
WHERE Name LIKE 'Ring%'
```

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

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

```
SELECT COUNT(Id)
FROM Widget__c
WHERE Name LIKE 'Ring%'
```

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

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

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

```
SELECT COUNT(Id), COUNT(Widget__r.Id)
FROM Model__c
```

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

```
SELECT Widget__c, COUNT(Name)
FROM Model__c
GROUP BY Widget__c
```

Support for Field Types in Aggregate Functions

Aggregate functions provide a powerful means to analyze your records, but they are not relevant for all field types. For example, using an aggregate function on a base64 field type would not return meaningful data, so base64 fields do not support any of the aggregate functions. 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

Data Type	`AVG()`	`COUNT()`	`COUNT_DISTINCT()`	`MIN()`	`MAX()`	`SUM()`
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()`
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
email	No	Yes	Yes	Yes	Yes	No
encryptedstring	No	No	No	No	No	No
ID	No	Yes	Yes	Yes	Yes	No
masterrecord	No	Yes	Yes	Yes	Yes	No
multipicklist	No	No	No	No	No	No
percent	Yes	Yes	Yes	Yes	Yes	Yes
phone	No	Yes	Yes	Yes	Yes	No
picklist	No	Yes	Yes	Yes	Yes	No
reference	No	Yes	Yes	Yes	Yes	No
textarea	No	Yes	Yes	Yes	Yes	No
url	No	Yes	Yes	Yes	Yes	No

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

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

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

Date Functions

Date functions allow you to group or filter your data by various date periods. 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(Widget_Cost__c)
FROM Widget__c
GROUP BY CALENDAR_YEAR(CreatedDate)
```

Date functions are available in API version 18.0 and later.

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

This table lists all the date functions supported by SOQL.

Date Function	Description	Examples
`CALENDAR_MONTH()`	Returns a number representing the calendar month of a date field.	• `1` for January • `12` for December
`CALENDAR_QUARTER()`	Returns a number representing the calendar quarter of a date field.	• `1` for January 1 through March 31 • `2` for April 1 through June 30 • `3` for July 1 through September 30 • `4` for October 1 through December 31
`CALENDAR_YEAR()`	Returns a number representing the calendar year of a date field.	`2009`
`DAY_IN_MONTH()`	Returns a number representing the day in the month of a date field.	`20` for February 20
`DAY_IN_WEEK()`	Returns a number representing the day of the week for a date field.	• `1` for Sunday • `7` for Saturday
`DAY_IN_YEAR()`	Returns a number representing the day in the year for a date field.	`32` for February 1
`DAY_ONLY()`	Returns a date representing the day portion of a dateTime field.	`2009-09-22` for September 22, 2009 You can only use `DAY_ONLY()` with dateTime fields.
`FISCAL_MONTH()`	Returns a number representing the fiscal month of a date field. This differs from `CALENDAR_MONTH()` if your organization uses a fiscal year that does not match the Gregorian calendar. **Note:** This function is not supported if your organization has custom fiscal years enabled. See "About Fiscal Years" in the Database.com online help.	If your fiscal year starts in March: • `1` for March • `12` for February See “Setting the Fiscal Year” in the Database.com online help.

Date Function	Description	Examples
`FISCAL_QUARTER()`	Returns a number representing the fiscal quarter of a date field. This differs from `CALENDAR_QUARTER()` if your organization uses a fiscal year that does not match the Gregorian calendar. **Note:** This function is not supported if your organization has custom fiscal years enabled. See "About Fiscal Years" in the Database.com online help.	If your fiscal year starts in July: • `1` for July 15 • `4` for June 6
`FISCAL_YEAR()`	Returns a number representing the fiscal year of a date field. This differs from `CALENDAR_YEAR()` if your organization uses a fiscal year that does not match the Gregorian calendar. **Note:** This function is not supported if your organization has custom fiscal years enabled. See "About Fiscal Years" in the Database.com online help.	`2009`
`HOUR_IN_DAY()`	Returns a number representing the hour in the day for a dateTime field.	`18` for a time of 18:23:10 You can only use `HOUR_IN_DAY()` with dateTime fields.
`WEEK_IN_MONTH()`	Returns a number representing the week in the month for a date field.	`2` for April 10 The first week is from the first through the seventh day of the month.
`WEEK_IN_YEAR()`	Returns a number representing the week in the year for a date field.	`1` for January 3 The first week is from January 1 through January 7.

Note the following when you use date functions:

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

  ```
  SELECT CreatedDate, Widget_Cost__c
  FROM Widget__c
  WHERE CALENDAR_YEAR(CreatedDate) = 2011
  ```

- 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, Widget_Cost__c
  FROM Widget__c
  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), Widget_Cost__c
FROM Widget__c
```

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(CreateDate)
FROM Widget__c
GROUP BY CALENDAR_YEAR(CreateDate)
```

Converting Time Zones in Date Functions

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

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

```
SELECT HOUR_IN_DAY(convertTimezone(CreatedDate)), SUM(Widget_Cost__c)
FROM Widget__c
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 Widget__c
```

Querying Currency Fields in Multicurrency Organizations

If an organization is multicurrency enabled, you can use convertCurrency() in the SELECT clause to convert currency fields to the user's currency.

Use this syntax for the SELECT clause:

```
convertCurrency(field)
```

For example:

```
SELECT Id, convertCurrency(Widget_Cost__c)
FROM Widget__c
```

If an organization has enabled advanced currency management, dated exchange rates will be used when converting currency fields on opportunities, opportunity line items, and opportunity history.

You cannot 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 organization:

```
WHERE Object_name Operator ISO_CODEvalue
```

For example:

```
SELECT Id, Name
FROM Widget__c
WHERE Widget_Cost__c > USD5
```

In this example, Widget records will be returned if the record's currency `Widget_Cost__c` value is greater than the equivalent of USD5. For example, a Widget with an amount of `USD6` would be returned, but not `JPY100`.

Use an ISO code that your organization has enabled and is active. If you do not put in an ISO code, then the numeric value is used instead of comparative amounts. Using the example above, opportunity records with `JPY5001`, `EUR5001`, and `USD5001` would be returned. Note that if you use IN in a `WHERE` clause, you cannot mix ISO code and non-ISO code values.

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

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

For example:

```
SELECT Name, MAX(Widget_Cost__c)
FROM Widget__c
GROUP BY Name
HAVING MAX(Widget_Cost__c) > 10
```

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 does not work:

```
SELECT Name, MAX(Widget_Cost__c)
FROM Widget__c
GROUP BY Name
HAVING MAX(Widget_Cost__c) > USD10
```

Example `SELECT` Clauses

Type of Search	Example(s)
Simple query	`SELECT Id, Name FROM Widget__c`
`WHERE`	`SELECT Id FROM Model__c WHERE Name LIKE 'W%'`
`ORDER BY`	`SELECT Name FROM Widget__c ORDER BY Name DESC NULLS LAST`
`LIMIT`	`SELECT Name FROM Model__c WHERE Widget__r.Name = 'Ring' LIMIT 12`
`ORDER BY` with `LIMIT`	`SELECT Name FROM Model__c WHERE Widget__r.Name = 'Ring' ORDER BY CreatedDate ASC NULLS LAST LIMIT 12`
`count()`	`SELECT COUNT() FROM Model__c`
`GROUP BY`	`SELECT Widget__r.Name, COUNT(Name) FROM Model__c GROUP BY Widget__r.Name`

Type of Search	Example(s)
`HAVING`	`SELECT Name, COUNT(Id) FROM Widget__c GROUP BY Name HAVING COUNT(Id) > 1`
`OFFSET` with `ORDER BY`	`SELECT Name, Id FROM Merchandise__c ORDER BY Name OFFSET 100`
`OFFSET` with `ORDER BY` and `LIMIT`	`SELECT Name, Id FROM Merchandise__c ORDER BY Name LIMIT 20 OFFSET 100`

Relationship queries: child-to-parent	`SELECT Model__c.Name, Model__c.Widget__r.Name FROM Model__c`
	
	`SELECT Id, Name, Widget__r.Name FROM Model__c WHERE Widget__r.Widget_Cost__c > 2.00`

Relationship queries: parent-to-child	`SELECT Name, (SELECT Name FROM Models__r) FROM Widget__c`
	
	`SELECT Widget__c.Name, (SELECT Model__c.Name FROM Widget__c.Models__r) FROM Widget__c`

Relationship query with `WHERE`	`SELECT Name, (SELECT Name FROM Models__r WHERE CreatedBy.Alias = 'x') FROM Widget__c WHERE Name = 'Ring'`
Polymorphic relationship queries using `TYPEOF`	`SELECT TYPEOF Owner WHEN User THEN FirstName, LastName WHEN Group THEN Name, Email ELSE Name END FROM Merchandise__c` **Note:** `TYPEOF` is currently available as a Developer Preview as part of the SOQL Polymorphism feature. For more information on enabling `TYPEOF` for your organization, contact salesforce.com.
Relationship queries with aggregate	`SELECT Name, (SELECT CreatedBy.Name FROM Models__r) FROM Widget__c`
Simple query: the UserId and LoginTime for each user	`SELECT UserId, LoginTime from LoginHistory`
Relationship queries with number of logins per user in a specific time range	`SELECT UserId, COUNT(Id) from LoginHistory WHERE LoginTime > 2010-09-20T22:16:30.000Z AND LoginTime < 2010-09-21T22:16:30.000Z GROUP BY UserId`

Note: Apex requires that you surround SOQL and SOSL statements with square brackets in order to use them on the fly. Additionally, Apex script variables and expressions can be used if preceded by a colon (`:`).

Relationship Queries

Client applications need to be able to query for more than a single type of object at a time. SOQL provides syntax to support these types of queries, called *relationship queries*, against both standard objects and custom objects.

Relationship queries traverse parent-to-child and child-to-parent relationships between objects to filter and return results. They 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.

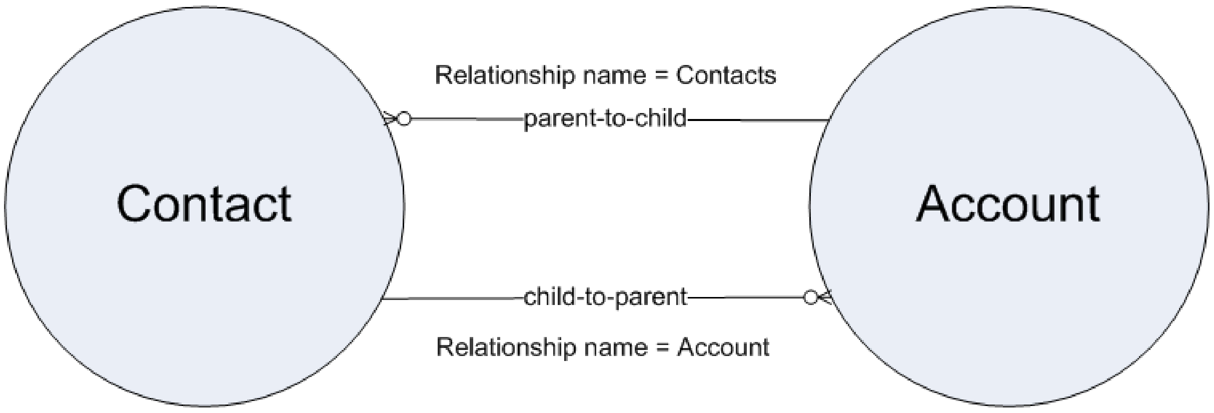
You can use relationship queries to return objects of one type based on criteria that applies to objects of another type, for example, “return all accounts created by Bob Jones and the contacts associated with those accounts.” There must be a parent-to-child or child-to-parent relationship connecting 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
- Lookup Relationships and Outer Joins
- Identifying Parent and Child Relationships
- Understanding Polymorphic Keys and Relationships
- Understanding Relationship Query Limitations
- Using Relationship Queries with History 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 Model child object has a child-to-parent relationship to the Widget object, so the value of `relationshipName` in Contact is `Widget__r`. These relationships are traversed by specifying the parent using dot notation in the query, for example:

  ```
  SELECT Model__c.Name, Model__c.Widget__r.Name from Model__c
  ```

  This query returns the names of all the Models in the organization, and for each Model, the Widget name associated with (parent of) that Model.

- 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, Widget has a child relationship to Models and has a `relationshipName` for them called `Models`. These relationships can be traversed only in the `SELECT` clause, using a nested SOQL query. For example:

  ```
  SELECT Widget__c.Name, (SELECT Model__c.Name FROM Widget__c.Models__r) FROM Widget__c
  ```

This query returns all Widgets, and for each Widget, the name of each Model associated with (the child of) that Widget.

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

Relationship names are somewhat different for custom objects, though the `SELECT` syntax is the same. See Understanding Relationship Names, Custom Objects, and Custom Fields for more information.

Using Relationship Queries

You can query the following relationships using SOQL:

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

  For example:

  ```
  SELECT Id, Name, Widget__r.Name
  FROM Model__c
  WHERE Widget__r.CreatedBy.LastName LIKE 'Smi%'
  ```

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

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

  For example:

  ```
  SELECT Name,
    (
      SELECT Name
      FROM Models__r
    )
  FROM Widget__c
  ```

  The query returns the name for all the Widgets, and for each Widget, the name of each Model.

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

  For example:

  ```
  SELECT Name,
    (
      SELECT CreatedBy.Name
      FROM Models__r
    )
  FROM Widget__c
  ```

  This query returns Widget__c records, and for each widget, the name of the widget, the Model__c records for those widgets (which can be an empty result set if there were no Model__c records on any widget) with the name of the user who created each Model__c (if the result set is not empty).

- Any query (including subqueries) can include a `WHERE` clause, which applies to the object in the `FROM` clause of the current query. These clauses can filter on any object in the current scope (reachable from the root element of the query), via the parent relationships.

  For example:

  ```
  SELECT Name,
    (
      SELECT Name
      FROM Models__r
      WHERE CreatedBy.Alias = 'x'
    )
  FROM Widget__c WHERE Name = 'Ring'
  ```

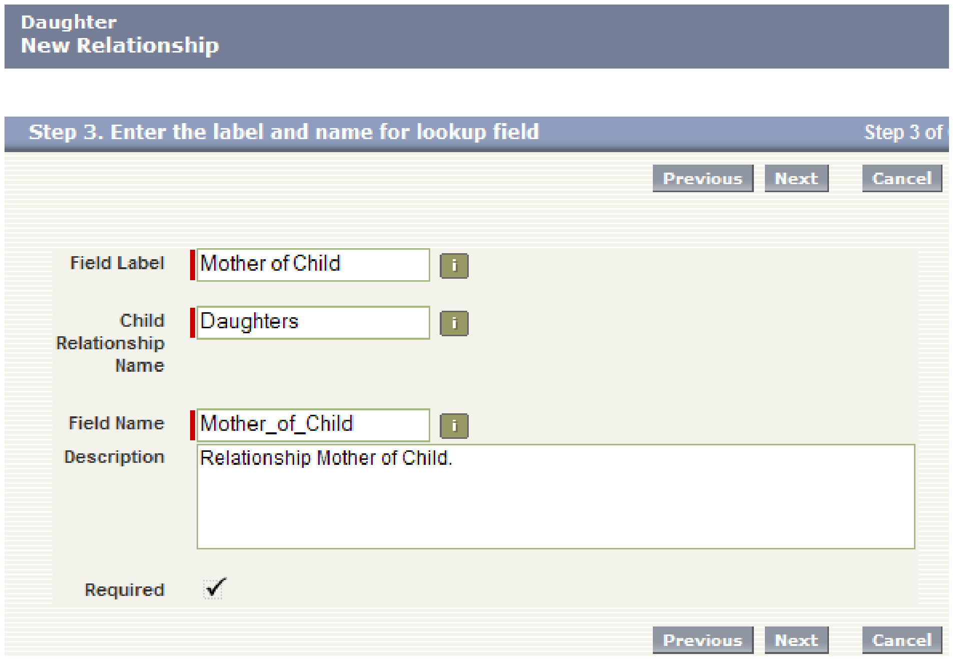
  This query returns the name for all Widgets whose name is 'Ring', and for each Widget returned, returns the name of every Model whose created-by alias is 'x.'

Understanding Relationship Names, Custom Objects, and Custom Fields

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

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

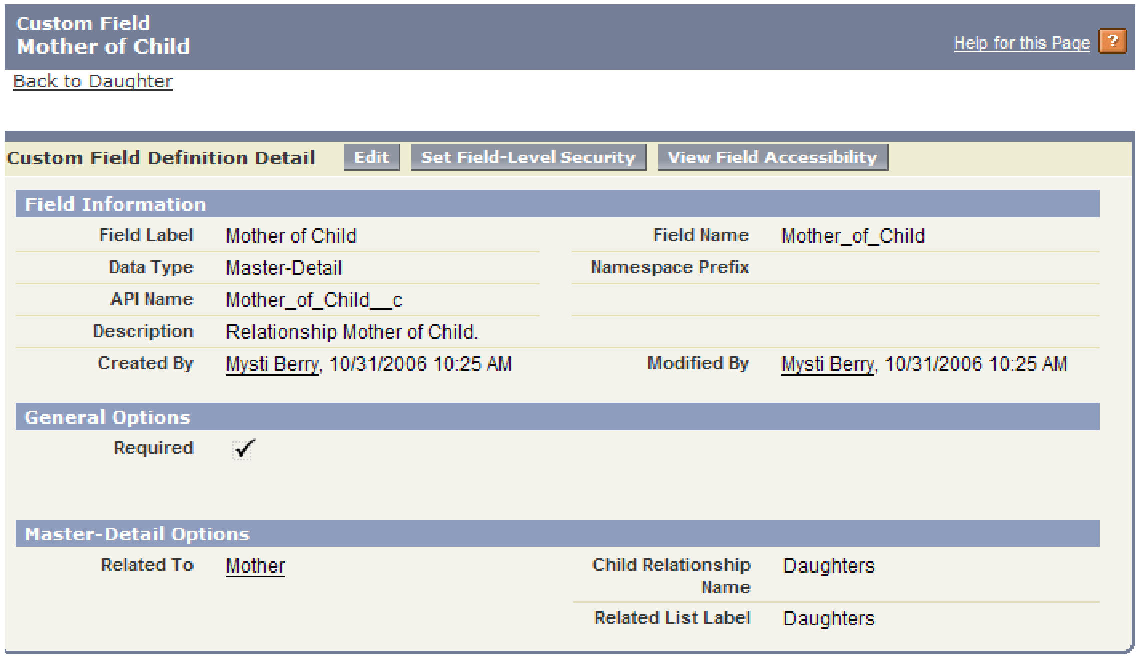
When you create a new custom relationship in the Database.com user interface, you are asked to specify the plural version of the object name, which you use for relationship queries:



Field Label	Mother of Child
Child Relationship Name	Daughters
Field Name	Mother_of_Child
Description	Relationship Mother of Child.
Required	✓

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):



Custom Field
Mother of Child

Custom Field Definition Detail

Field Information

Field Label	Mother of Child	Field Name	Mother_of_Child
Data Type	Master-Detail	Namespace Prefix	
API Name	Mother_of_Child__c		
Description	Relationship Mother of Child.		
Created By	Mysti Berry, 10/31/2006 10:25 AM	Modified By	Mysti Berry, 10/31/2006 10:25 AM

General Options

Required	✓

Master-Detail Options

Related To	Mother	Child Relationship Name	Daughters
		Related List Label	Daughters

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 Database.com 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` 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, Name, Widget__r.Id, Widget__r.Name
  FROM Model__c
  WHERE Widget__r.Name LIKE 'Ring%'
  ```

  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, Name
      FROM Models__r
    )
  FROM Widget__c
    WHERE Name like 'Ring%'
  ```

  This query returns a set of Widgets, and within each Widget, a query result set of Model fields containing the Model information from the subquery.

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

1. Perform the query on Widget__c.
2. Iterate over the Widget__c QueryResult with `queryMore()`.
3. For each Widget__c object, retrieve the Model__c QueryResult.
4. Iterate over the child models, using `queryMore()` on each model's QueryResult.

The following sample illustrates how to process subquery results:

```
private void querySample() {
  QueryResult qr = null;
  try {
    qr = connection.query("SELECT w.Id, w.Name, " +
      "(SELECT m.Id, m.Name FROM a.Models__r m) +
      "FROM Widget__c w");
    boolean done = false;
    if (qr.getSize() > 0) {
      while (!done) {
        for (int i = 0; i < qr.getRecords().length; i++) {
        Widget__c widget = (Widget__c) qr.getRecords()[i];
        String name = widget.getName();
        System.out.println("Widget " + (i + 1) + ": " + name);
        printModels(widget.getModels__r());
        }
        if (qr.isDone()) {
          done = true;
        } else {
          qr = connection.queryMore(qr.getQueryLocator());
        }
      }
    } else {
      System.out.println("No records found.");
    }
    System.out.println("\nQuery succesfully executed.");
  } catch (ConnectionException ce) {
    System.out.println("\nFailed to execute query successfully, error message " +
    "was: \n" + ce.getMessage());
```

```
  }
}

private void printModels(QueryResult qr) throws ConnectionException {
  boolean done = false;
  if (qr.getSize() > 0) {
    while (!done) {
    for (int i = 0; i < qr.getRecords().length; i++) {
      Model__c model = (Model__c) qr.getRecords()[i];
      String name = model.getName();
      System.out.println("Child model " + (i + 1) + ": " + name );
    }
    if (qr.isDone()) {
      done = true;
    } else {
      qr = connection.queryMore(qr.getQueryLocator());
    }
    }
  } else {
    System.out.println("No child records found.");
  }
}
```

Lookup Relationships and Outer Joins

Beginning with version 13.0 of the API, relationship queries return records even if the relevant foreign key field has a null value, as you would expect with an outer join. The change in behavior applies to the following types of relationship queries:

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

  ```
  SELECT Id, Name, Widget__r.Id, Widget__r.Name
  FROM Model__c
  ORDER BY Widget__r.Name
  ```

  Any Model record for which `Widget__c` is empty is returned in version 13.0 and later.

- In a WHERE clause using `OR`, if the foreign key value in a record is null, the record is returned in version 13.0 and later, but not returned in versions before 13.0. For example, if your organization has one Model with the value of its `Name` field equal to `Wallace` and the value of its `Widget__r` field equal to `null`, and another Model with a different name and a parent Widget named `Ring`, the following query returns only the Model with the name equal to `Ring`:

  ```
  SELECT Id FROM Model__c WHERE Name = 'Wallace' OR Widget__r.Name = 'Ring'
  ```

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

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

  ```
  SELECT Id
  FROM Model__c
  WHERE Widget__r.name = null
  ```

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

Identifying Parent and Child Relationships

You can identify parent-child relationships by viewing the ERD diagrams in the Data Model section of the *Database.com 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 Widget__c:

```
<complexType name="Widget__c">
  <complexContent>
    <extension base="ens:sObject">
      <sequence>
      ...
        <element name="Models__r" nillable="true" minOccurs="0"
                 type="tns:QueryResult"/>
      ...
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

  In the example above, the child relationship name `Models` is in the entry for its parent Widget.

- For the parent of an object, look for a pair of entries, such as `Widget__c` and `Widget__r`, where the ID field represents the parent object referenced by the ID, and the other represents the contents of the record.

```
<complexType name="Model__c">
  <complexContent>
    <extension base="ens:sObject">
      <sequence>
        ...
        <element name="Widget__c" nillable="true" minOccurs="0"
                 type="tns:ID"/>
        <element name="Widget__r" nillable="true" minOccurs="0"
                 type="ens:Widget__c"/>
      </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.

Understanding Polymorphic Keys and Relationships

In a polymorphic relationship, the referenced object of the relationship can be one of several different objects. For example, the Owner relationship field of a Merchandise__c custom object could be a User or a Group. When making queries or updating records with polymorphic relationships, you need to check the actual object type set for the relationship, and act accordingly. You can access polymorphic relationships several ways.

- You can use the polymorphic key for the relationship.
- You can use a `TYPEOF` clause in a query.

- You can use the Type qualifier on a polymorphic field.

You can also combine these techniques for complex queries. Each of these techniques are described below.

Note: `TYPEOF` is currently available as a Developer Preview as part of the SOQL Polymorphism feature. For more information on enabling `TYPEOF` for your organization, contact salesforce.com.

Using Polymorphic Keys

A polymorphic key is an ID that can refer to more than one type of object as a parent. For example, either a User or a Group can be the owner of a custom object. In other words, the `OwnerId` field of a custom object can contain the ID of either a User or a Group. If an object can have more than one type of object as a parent, the polymorphic key points to a Name object instead of a single object type.

Executing a `describeSObjects()` call returns the Name object, whose field `Type` contains a list of the possible object types that can parent the queried object. The `namePointing` field in the DescribeSObjectResult indicates that the relationship points to the Name object, needed because the relationship is polymorphic. For example, the value in the `OwnerId` field in a Merchandise__c record can be a User or Group.

Note: If your organization has the SOQL Polymorphism feature enabled, polymorphic relationship fields reference sObjects, and not Names.

In order to traverse relationships where the object type of the parent is not known, you can use these fields to construct a query:

- `owner`: This field represents the object of a parent who owns the child object, regardless of the parent's object type. For example:

  ```
  SELECT Id, Owner.Name
  FROM Merchandise__c
  WHERE Owner.Name like 'R%'
  ```

  This example query works for Merchandise__c records whose owners are either Users or Groups.

You can also use `describeSObjects()` to obtain information about the parents and children of objects. For more information, see `describeSObjects()` and especially `namePointing`, which, if set to `true`, indicates the field points to a name.

Using `TYPEOF`

SOQL supports polymorphic relationships using the `TYPEOF` expression in a `SELECT` statement. `TYPEOF` is available in API version 26.0 and later.

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 Owner polymorphic relationship field in a Merchandise__c record.

```
SELECT
  TYPEOF Owner
    WHEN User THEN FirstName, LastName
    WHEN Group THEN Name, Email
    ELSE Name
  END
FROM Merchandise__c
```

At run time this `SELECT` statement checks the object type referenced by the Owner field in a Merchandise__c. If the object type is User, the referenced User's FirstName and LastName fields are returned. If the object type is Group, the referenced Group's Name and Email fields are returned. If the object type is any other type, the Name field is returned. Note that if an `ELSE` clause isn't provided and the object type isn't User or Group,then `null` is returned for that Merchandise__c record.

Note the following considerations for `TYPEOF`.

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

```
SELECT Name From Line_Item__c
WHERE CreatedById IN
    (
    SELECT
        TYPEOF Owner
            WHEN User THEN Id
            WHEN Group THEN CreatedById
        END
    FROM Merchandise__c
    )
```

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

```
SELECT
    TYPEOF Owner
        WHEN User THEN FirstName
        WHEN Group THEN Name
    END
FROM Line_Item__c
WHERE CreatedById IN
    (
    SELECT CreatedById
    FROM Merchandise__c
    )
```

- GROUP BY and HAVING aren't allowed in queries that use TYPEOF.

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 Owner field in Merchandise__c.

```
SELECT Description
FROM Merchandise__c
WHERE Owner.Type IN ('User', 'Group')
```

At run time this SELECT statement returns the descriptions for Merchandise__c records that reference Users or Groups in the Owner field. If a Merchandise__c references something other than a User or Group in the Owner 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.

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 Owner field in Merchandise__c.

```
SELECT Id,
  TYPEOF Owner
    WHEN User THEN FirstName
    WHEN Group THEN Email
  END
FROM Merchandise__c
WHERE Owner.Type IN ('User', 'Group')
```

At run time this `SELECT` statement always returns the ID for a Merchandise__c, and then either User.FirstName, or Group.Email, depending on the object type referenced by the Merchandise__c record's Owner field. Note that no `ELSE` clause has been provided. Since the statement filters based on the Owner field in the `WHERE` clause, only Merchandise__c records that reference either a User or Group 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.

Understanding Relationship Query Limitations

When designing relationship queries, consider these limitations:

- Relationship queries are not the same as SQL joins. You must have a relationship between objects to create a join in SOQL.
- No more than 35 child-to-parent relationships can be specified in a query. A custom object allows up to 25 relationships, so you can reference all the child-to-parent relationships for a custom object in one query.
- No more than 20 parent-to-child relationships can be specified in a 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 each specified relationship, only one level of parent-to-child relationship can be specified in a query. For example, if the `FROM` clause specifies Account, the `SELECT` clause can only specify the Contact or other objects at that level. It could not specify a child object of Contact.

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 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 foo__history)
FROM foo__c
```

Using Relationship Queries with the Partner WSDL

The partner WSDL does not contain the detailed type information available in the enterprise WSDL to get the information you need for a relationship 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 a Widget object, the relationship name for the Model child is `Models__r`.
- The reference fields available for the relevant object.

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

Changing the Batch Size in Queries

By default, the number of rows returned in the query result object (batch size) returned in a `query()` or `queryMore()` call is set to 500. WSC clients can set the batch size by calling `setQueryOptions()` on the connection object. C# client applications can change this setting by specifying the batch size in the call QueryOptions portion of the SOAP header before invoking the `query()` call. The maximum batch size is 2,000 records. However this setting is only a suggestion. There is no guarantee that the requested batch size will be the actual batch size. This is done to maximize performance.

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

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

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

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

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

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

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, as well as to traverse relationships between objects. Several options can also 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 Database.com online help for syndication feeds.

Location-Based SOQL Queries—Beta

Location-based SOQL queries let you compare and query location values stored in Database.com. You can calculate the distance between two location values (such as between a warehouse and a store), or between a location value and any 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. Geolocation fields identify a location by its latitude and longitude. Additionally, standard addresses on Database.com objects include a geolocation field and, once populated, can be used in similar ways (with a few restrictions). Locations of both types can be compared and queried, for example, to find the 10 closest accounts.

See “Compound Fields—Beta” in the *SOAP API Developer’s Guide* for additional details of using address and geolocation compound fields, including a few limitations of the beta.

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. And, if the geolocation field of a standard address has been populated by a geocoding service, latitude and longitude values can be used 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 of the warehouses that are stored in the custom object Warehouse. The results include each warehouse’s latitude and longitude values individually. Note the use of the separate field components for `Location__c`, to select the latitude and longitude components individually.

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. Results from a REST API request might look something like this.

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

WHERE Clause

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

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 or a location field and a value returned by the GEOLOCATION function. Replace *`unit`* with mi (miles) or km (kilometers).

GEOLOCATION

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

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

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

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

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 will be first in the list; the farthest location will be 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.

Records that have invalid geolocation field values, that is, geolocations where either latitude or longitude is null, but not both, are treated as though both values are null when used in SOQL `WHERE DISTANCE()` and `ORDER BY` clauses. In other words, 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
```

The values returned from an API call might look something like these values.

Name	Location__c
Ferry Building Depot	*null*
Aloha Warehouse	(37.786108,-122.430152)

Name	Location__c
Big Tech Warehouse	*null*
S H Frank & Company	*null*
San Francisco Tech Mart	(37.77587,-122.399902)

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

When the individual field components of that same geolocation field are used in a `SELECT` clause, the saved values are returned as before. In other words, non-null values will be 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
```

In this example, the results might look like these values.

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

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

Limitations on Location-Based SOQL Queries

As a beta-release feature, location-based queries are supported in SOQL in Apex and in the SOAP and REST APIs with the following limitations:

- Outside of the SOAP and REST APIs, geolocation fields are supported in the SELECT clause of SOQL queries only at the component level. In other words, you have to query the latitude or longitude; you can't query the compound location field. Specify geolocation field components by appending "__latitude__s" or "__longitude__s" to the field name, instead of the usual "__c".
- DISTANCE and GEOLOCATION are supported in WHERE and ORDER BY clauses in SOQL, but not in GROUP BY or SELECT.
- DISTANCE only supports the logical operators > and <, returning values within (<) or beyond (>) a specified radius.
- Syntax is restricted when running SOQL queries: When using the GEOLOCATION function, 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.

See "Compound Field Considerations and Limitations" in the *SOAP API Developer's Guide* for additional details.

Chapter 2

Salesforce Object Search Language (SOSL)

In this chapter ...

Use the Salesforce Object Search Language (SOSL) to construct text searches in the following environments:

- the `search()` call
- Apex statements
- the Schema Explorer of the Eclipse Toolkit

Unlike SOQL, which can only query one object at a time, SOSL enables you to search text, email, and phone fields for multiple objects simultaneously.

About SOSL

SOSL allows you to specify the following for source objects:

- text expression
- scope of fields to search
- list of objects and fields to retrieve
- conditions for selecting rows in the source objects

Pass the entire SOSL expression in the `search` parameter of the `search()` call.

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

Comparing SOSL and SOQL

Like Salesforce Object Query Language (SOQL), SOSL allows you to search your organization's Database.com data for specific information. Unlike SOQL, which can only query one object at a time, a single SOSL query can search all objects—including custom objects—to which you have access. The API executes the search within the specified scope and returns to you only the information that is available to you based on the user permissions under which your application has logged in.

- Use SOQL with the `query()` call to select records for a single object.
- Use SOSL with the `search()` call to find records for one or more objects. The `search()` call searches most text fields on an object. See Search Scope for information on the fields searched.

Designing Efficient Text Searches

If your searches are too general, they will be slow and return too many results. Use the following to write more efficient searches:

- IN clause—for limiting the types of columns to search
- RETURNING clause—for limiting the objects to search
- LIMIT clause—for restricting the search results
- OFFSET clause—for paging the search results

Search Scope

The `search()` call searches most objects (including custom objects) and text fields to which you have access. It does not search the following objects and fields:

- Any elements such as picklists that are defined as not searchable (`searchable` is `false`). To determine whether a given object is searchable, your application can invoke the `describeSObjects()` call on the object and inspect the `searchable` property in the DescribeSObjectResult.
- Number, date, or checkbox fields. To search for such information, use the `query()` call instead.
- Textarea fields, unless you use the `ALL FIELDS` search group.

Note the following implementation tips:

- In most cases, the `search()` call does not provide specialized search features such as synonym matching or stop words.
- Apex requires that you surround SOQL and SOSL statements with square brackets in order to use them on the fly. Additionally, Apex script variables and expressions can be used if preceded by a colon (`:`).

SOSL Typographical Conventions

Topics about SOSL use the following typographical conventions:

Convention	Description
`FIND Name IN Widget__c`	In an example, 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 question marks and square brackets.
`FIND` ***`fieldname`*** `IN` ***`objectname`***	In an example or syntax statement, italics represent variables. You supply the actual value.
`,`	In a syntax statement, the comma inside square brackets indicates that the element containing it may be repeated up to the limits for that element.
`[ORDER BY` ***`conditionexpression`***`]`	In a syntax statement, square brackets surround an element that is optional. You may omit the element, or include one, or if a comma is present, more than one of them.

SOSL Syntax

The SOSL query syntax consists of a required `FIND` clause followed by one or more optional clauses in the following order:

```
FIND {SearchQuery} [ IN SearchGroup [ convertCurrency(Amount)] ]
[ RETURNING FieldSpec ]
[ LIMIT n ]
[ UPDATE TRACKING ]
[ UPDATE VIEWSTAT ]
```

where:

Syntax	Description
`FIND` *`{SearchQuery}`*	Required. Specifies the text (words or phrases) to search for. The search query must be delimited with curly braces. If the `SearchQuery` string is longer than 10,000 characters, no result rows are returned. If `SearchQuery` is longer than 4,000 characters, any logical operators are removed. For example, the `AND` operator in a statement with a `SearchQuery` that's 4,001 characters will default to the `OR` operator, which could return more results than expected.
`IN` *`SearchGroup`*	Optional. Scope of fields to search. One of the following values: • `ALL FIELDS` • `NAME FIELDS` • `EMAIL FIELDS` • `PHONE FIELDS`

Syntax	Description
	If unspecified, then the default is `ALL FIELDS`. You can specify the list of objects to search in the `RETURNING` *`FieldSpec`* clause. **Note:** This clause doesn't apply to feed comments, feed items, and files. If any of these objects are specified in the RETURNING clause, the search is not limited to specific fields; all fields are returned.
`convertCurrency(`***`Amount`***`)`	Optional. If an organization is multicurrency enabled, converts currency fields to the user's currency.
`RETURNING` *`FieldSpec`*	Optional. Information to return in the search result. List of one or more objects and, within each object, list of one or more fields, with optional values to filter against. If unspecified, then the search results contain the IDs of all objects found.
`LIMIT` *`n`*	Optional. Specifies the maximum number of rows returned 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 returned.
`UPDATE TRACKING`	Optional. If an organization uses Salesforce Knowledge, tracks keywords used in Salesforce Knowledge article search.
`UPDATE VIEWSTAT`	Optional. If an organization uses Salesforce Knowledge, updates an article's view statistics.

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

FIND *{SearchQuery}*

The required `FIND` clause allows you to specify the word or phrase to search for. 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, searching for `Customer`, `customer`, or `CUSTOMER` all return the same results.

Note that special types of text expressions (such as macros, functions, or regular expressions) that are evaluated at run time are not allowed in the `FIND` clause.

Note: The `SearchQuery` must be delimited with curly braces. This is needed to unambiguously distinguish the search expression from other clauses in the text search.

Single Words and Phrases

A `SearchQuery` contains two types of text:

- **Single Word**— single word, such as `test` or `hello`. Words in the `SearchQuery` are delimited by spaces, punctuation, and changes from letters to digits (and vice-versa). Words are always case insensitive. In Chinese, Japanese, and Korean (CJK), words are also delimited by pairs of CJK-type characters.
- **Phrase**— collection of words and spaces surrounded by double quotes such as `"john smith"`. Multiple words can be combined together with logical and grouping Operators to form a more complex query. Certain keywords ("and," "or," and "and not") must be surrounded in double quotes if you want to search for those words, otherwise they are interpreted as the corresponding operator.

Wildcards

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

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

When using wildcards, consider the following issues:

- Wildcards take on the type of the preceding character. For example, `aa*a` matches *aaaa* and *aabcda*, but not *aa2a* or *aa.!//a*, and `p?n` matches *pin* and *pan*, but not *p1n* or *p!n*. Likewise, `1?3` matches *123* and *143*, but not *1a3* or *1b3*.
- A wildcard (*) is appended at the end of single characters in Chinese, Japanese, Korean, and Thai (CJKT) searches, except in exact phrase searches.
- 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

You can use the following special operators to focus your text search. Operator support is case-insensitive.

Operator	Description
`" "`	Use quotation marks around search terms to find an exact phrase match. This can be especially useful when searching for text with punctuation. For example, `"acme.com"` finds items that contain the exact text `acme.com`. A search for `"monday meeting"` finds items that contain the exact phrase `monday meeting`.
`AND`	Finds items that match all of the search terms. For example, `john AND smith` finds items with both the word `john` and the word `smith`. In most cases if an operator isn't specified, `AND` is the default operator. Case-insensitive.

Operator	Description
`OR`	Finds items with at least one of the search terms. For example, `john OR smith` finds items with either `john` or `smith`, or both words. Case-insensitive.
`AND NOT`	Finds items that do not contain the search term. For example, `john AND NOT smith` finds items that have the word `john` but not the word `smith`. Case-insensitive.
( )	Use parentheses around search terms in conjunction with logical operators to group search terms. For example, you can search for: • `("Bob" and "Jones") OR ("Sally" and "Smith")`—searches for either Bob Jones or Sally Smith. • `("Bob") and ("Jones" OR "Thomas") and Sally Smith`—searches for documents that contain Bob Jones and Sally Smith or Bob Thomas and Sally Smith.

`SearchQuery` Character Limits

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

Search Order

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

1. () (parentheses)
2. AND and AND NOT (evaluated from right to left)
3. OR

These examples show how search strings are evaluated:

Searching for...	Is the same as...	Finds items with the words...
`acme AND california AND NOT meeting`	`acme AND (california AND NOT meeting)`	*acme* and *california* but not *meeting*
`acme AND NOT california AND meeting`	`acme AND NOT (california AND meeting)`	*acme* but not with both *california* and *meeting*
`acme AND california OR meeting`	`(acme AND california) OR meeting`	*acme* and *california* and items with the word *meeting*
`acme AND (california OR meeting)`	`acme AND (california OR meeting)`	*acme* and *california* and items with the words *acme* and *meeting*

Reserved Characters

The following characters are reserved:

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

Reserved characters, if specified in a text search, must be escaped (preceded by the backslash \ character) in order 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	Example(s)

Single term examples	`FIND {MyProspect}`
	`FIND {mylogin@mycompany.com}`
	`FIND {FIND}`
	`FIND {IN}`
	`FIND {RETURNING}`
	`FIND {LIMIT}`

Single phrase	`FIND {John Smith}`
Term OR Term	`FIND {MyProspect OR MyCompany}`
Term AND Term	`FIND {MyProspect AND MyCompany}`
Term AND Phrase	`FIND {MyProspect AND "John Smith"}`
Term OR Phrase	`FIND {MyProspect OR "John Smith"}`

Complex query using AND/OR	`FIND {MyProspect AND "John Smith" OR MyCompany}`
	`FIND {MyProspect AND ("John Smith" OR MyCompany)}`

Complex query using AND NOT	`FIND {MyProspect AND NOT MyCompany}`
Wildcard search	`FIND {My*}`
Escape sequences	`FIND {Why not\?}`
Invalid or incomplete phrase (will not succeed)	`FIND {"John Smith}`

FIND Clauses in Apex

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

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

  ```
  FIND 'map*' IN ALL FIELDS RETURNING Merchandise__c (Id, Name), Invoice_Statement__c,
  Line_Item__c
  ```

- In the Force.com API, the value of the `FIND` clause is demarcated with braces. For example:

  ```
  FIND {map*} IN ALL FIELDS RETURNING Merchandise__c (Id, Name), Invoice_Statement__c,
  Line_Item__c
  ```

The *Database.com Apex Code Developer's Guide* has more information about using SOSL and SOQL with Apex.

IN *SearchGroup*

The optional `IN` clause allows you to define the types of fields to search. You can specify one of the following values (note that numeric fields are not searchable). If unspecified, the default behavior is to search all text fields in searchable objects.

Note: This clause doesn't apply to feed comments, feed items, and files. If any of these objects are specified in the RETURNING clause, the search is not limited to specific fields; all fields are returned.

Valid SearchGroup Settings

Scope	Description
`ALL FIELDS`	Search all searchable fields. If the `IN` clause is unspecified, then this is the default setting.
`EMAIL FIELDS`	Search only email fields.
`NAME FIELDS`	Search only name fields. In custom objects, fields that are defined as "Name Field" are searched. In custom objects, name fields have the `nameField` property set to `true`. (See the Field array of the `fields` parameter of the DescribeSObjectResult for more information.)
`PHONE FIELDS`	Search only phone number fields.

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

Example IN Clauses

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

RETURNING *FieldSpec*

The optional `RETURNING` clause allows you to specify the information that is returned in the text search result. If unspecified, then the default behavior is to return the IDs of all objects that are searchable up to the maximum specified in the `LIMIT n` clause or 2,000 (API version 28.0 and later), whichever is smaller. In the results, objects are listed in the order specified in the clause. API version 27.0 and earlier support a maximum of 200 results.

Note: Feed comments, feed items, and files must be specified explicitly in a `RETURNING` clause to be returned in search results. For example:

```
FIND {MyProspect} RETURNING FeedComment, FeedItem, ContentVersion
```

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 may be omitted. A comma indicates that the indicated segment can appear more than one time.

```
RETURNING ObjectTypeName
[(FieldList [WHERE conditionExpression] [ORDER BY clause] [LIMIT n] [OFFSET n])]
[, ObjectTypeName [(FieldList) [WHERE conditionExpression] [ORDER BY clause] [LIMIT n]
[OFFSET n])]]
```

RETURNING can contain the following elements:

Name	Description
ObjectTypeName	Object to return. If specified, then the search() call returns the IDs of all found objects matching the specified object. Must be a valid sObject type. You can specify multiple objects, separated by commas. Objects not specified in the RETURNING clause are not returned by the search() call.
FieldList	Optional list of one or more fields to return for a given object, separated by commas. If you specify one or more fields, the fields are returned for all found objects.

WHERE *conditionExpression*	Optional description of how search results for the given object should be filtered, based on individual field values. If unspecified, the search retrieves all the rows in the object that are visible to the user.
	
	Note that if you want to specify a WHERE clause, you must include a *FieldList* with at least one specified field. For example, this is not legal syntax:
	
	`RETURNING Widget__c (WHERE name like 'test')`
	
	But this is:
	
	`RETURNING Widget__c (Name, Widget_Cost__c WHERE Name like 'test')`
	
	See *conditionExpression* for more information.

ORDER BY *clause*	Optional description of how to order the returned result, including ascending and descending order, and how nulls are ordered. You can supply more than one ORDER BY clause.
	
	Note that if you want to specify an ORDER BY clause, you must include a *FieldList* with at least one specified field. For example, this is not legal syntax:
	
	`RETURNING Widget__c (ORDER BY id)`
	
	But this is:
	
	`RETURNING Widget__c (Name, Widget_Cost__c ORDER BY Name)`

Name	Description

`LIMIT` *n*	Optional clause that sets the maximum number of records returned for the given object. If unspecified, all matching records are returned, up to the limit set for the query as a whole.
	
	Note that if you want to specify a `LIMIT` clause, you must include a *`FieldList`* with at least one specified field. For example, this is not legal syntax:
	
	`RETURNING Widget__c (LIMIT 10)`
	
	But this is:
	
	`RETURNING Widget__c (Name, Widget_Cost__c LIMIT 10)`

`OFFSET` *n*	Optional clause used to specify the starting row offset into the result set returned by your query. `OFFSET` can be used only when querying a single object. `OFFSET` must be the last clause specified in a query.
	
	Note that if you want to specify an `OFFSET` clause, you must include a *`FieldList`* with at least one specified field. For example, this is not legal syntax:
	
	`RETURNING Widget__c (OFFSET 25)`
	
	But this is:
	
	`RETURNING Widget__c (Name, Widget_Cost__c OFFSET 25)`

Note: The `RETURNING` clause affects what data is returned, not what data is searched. The `IN` clause affects what data is searched.

Example RETURNING Clauses

Search Type	Example(s)
No Field Spec	`FIND {MyProspect}`
One object, no fields	`FIND {MyProspect} RETURNING Model__c`
Multiple objects, no fields	`FIND {MyProspect} RETURNING Model__c, Part__c`
One object, one or more fields	`FIND {MyProspect} RETURNING Widget__c(Name)`
	`FIND {MyProspect} RETURNING Model__c(Name, Id)`
Multiple objects, one or more fields, limits	`FIND {MyProspect} RETURNING Model__c(Name, Id LIMIT 10), Widget__c(Name, Widget_Cost__c)`
Multiple objects, mixed number of fields	`FIND {MyProspect} RETURNING Model__c(Name, Id), Widget__c, Part__c(Name)`
Single object limit	`FIND {MyProspect} RETURNING Model__c(Name, Id LIMIT 10)`
Multiple object limits and a query limit	`FIND {MyProspect} RETURNING Model__c(Name, Id LIMIT 20), Widget__c(Name, Widget_Cost__c LIMIT 10), Part__c LIMIT 50`
Single object offset	`FIND {MyProspect} RETURNING Model__c(Name, Id OFFSET 10)`

Note: Apex requires that you surround SOQL and SOSL statements with square brackets in order to use them on the fly. Additionally, Apex script variables and expressions can be used if preceded by a colon (`:`).

WHERE *conditionExpression*

The optional `WHERE` clause allows you to filter search results for a given object based on individual field values. If unspecified, the search retrieves all the rows in the object that are visible to the user.

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 Widget__c (id WHERE` **`createddate = THIS_FISCAL_QUARTER`**`)`
- `FIND {test} RETURNING Widget__c (id WHERE` **`cf__c includes('AAA')`**`)`
- You can use parentheses to define the order in which *`fieldExpression`*s are evaluated. For example, the following expression is `true` if `fieldExpression1` is `true` and either `fieldExpression2` or `fieldExpression3` are `true`:

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

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

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

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

fieldExpression

A *`fieldExpression`* uses the following syntax:

```
fieldName comparisonOperator value
```

where:

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

Comparison Operators

The following table lists the *comparisonOperator* values that are used in `fieldExpression` syntax. Note that comparisons on strings are case-insensitive.

Operator	Name	Description
`=`	`Equals`	Expression is true if the value in the specified *`fieldName`* equals the specified *`value`* in the expression. String comparisons using the equals operator are case-insensitive.
`!=`	`Not equals`	Expression is true if the value in the specified *`fieldName`* does not equal the specified *`value`*.
`<`	`Less than`	Expression is true if the value in the specified *`fieldName`* is less than the specified *`value`*.
`<=`	`Less or equal`	Expression is true if the value in the specified *`fieldName`* is less than, or equals, the specified *`value`*.
`>`	`Greater than`	Expression is true if the value in the specified *`fieldName`* is greater than the specified *`value`*.
`>=`	`Greater or equal`	Expression is true if the value in the specified *`fieldName`* is greater than or equal to the specified *`value`*.

`LIKE`	`Like`	Expression is true if the value in the specified *`fieldName`* matches the characters of the text string in the specified *`value`*. The `LIKE` operator in SOQL and SOSL is similar to the `LIKE` operator in SQL; it provides a mechanism for matching partial text strings and includes support for wildcards.
		• The `%` and _ wildcards are supported for the `LIKE` operator.
		• The `%` wildcard matches zero or more characters.
		• The _ wildcard matches exactly one character.
		• The text string in the specified *`value`* must be enclosed in single quotes.
		• The `LIKE` operator is supported for string fields only.
		• The `LIKE` operator performs a case-insensitive match, unlike the case-sensitive matching in SQL.
		• The `LIKE` operator in SOQL and SOSL supports escaping of special characters `%` or `_`.
		• Do not use the backslash character in a search except to escape a special character.
		
		For example, the following query matches Appleton, Apple, and Appl, but not Bappl:
		
		`SELECT Id, Name`
		`FROM Widget__c`
		`WHERE Name LIKE 'appl%'`

`IN`	`IN`	If the value equals any one of the specified values in a `WHERE` clause. For example:
		
		`SELECT Name FROM Widget__c`
		`WHERE Name IN ('Ring', 'Flange')`
		
		Note that the values for `IN` must be in parentheses. String values must be surrounded by single quotes.

Operator	Name	Description
		`IN` and `NOT IN` can also be used for semi-joins and anti-joins when querying on ID (primary key) or reference (foreign key) fields.
`NOT IN`	`NOT IN`	If the value does not equal any of the specified values in a `WHERE` clause. For example: `SELECT Name FROM Widget__c WHERE Name NOT IN ('Ring', 'Flange')` Note that the values for `NOT IN` must be in parentheses, and string values must be surrounded by single quotes. There is also a logical operator `NOT`, which is unrelated to this comparison operator.
`INCLUDES` `EXCLUDES`		Applies only to multi-select picklists.

Logical Operators

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

Operator	Syntax	Description
`AND`	***`fieldExpressionX`*** `AND` ***`fieldExpressionY`***	`true` if both *`fieldExpressionX`* and *`fieldExpressionY`* are true.
`OR`	***`fieldExpressionX`*** `OR` ***`fieldExpressionY`***	`true` if either *`fieldExpressionX`* or *`fieldExpressionY`* is true. Relationship queries with foreign key values in an `OR` clause behave differently depending on the version of the API. In a WHERE clause using `OR`, if the foreign key value in a record is null, the record is returned in version 13.0 and later, but not returned in versions before 13.0. `SELECT Id FROM Model__c WHERE Name = 'Wallace' OR Widget__r.Name = 'Ring'` The Model with no parent Widget has a name that meets the criteria, so it is returned in version 13.0 and later.
`NOT`	`not` ***`fieldExpressionX`***	`true` if *`fieldExpressionX`* is false. There is also a comparison operator `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

Sequence	Meaning
`\f` or `\F`	Form feed
`\"`	One double-quote character
`\'`	One single-quote character
`\\`	Backslash
LIKE expression only: `\_`	Matches a single underscore character (_)
LIKE expression only:`\%`	Matches a single percent sign character (%)

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

Example WHERE Clauses

Example(s)
`FIND {test} RETURNING Widget__c (id WHERE createddate = THIS_FISCAL_QUARTER)`
`FIND {test} RETURNING Widget__c (id WHERE cf__c includes('AAA'))`
`FIND {test} RETURNING Widget__c (id), Model__c(Field1,Field2 WHERE Field1 = 'test' order by id ASC, Name DESC)`
`FIND {test} IN ALL FIELDS RETURNING Widget__c(Name, Widget_Cost__c, id WHERE Name = 'test'), Model__c(Name, id), Part__c(id WHERE Part_Loc__c IN ('California', 'New York'))`
`FIND {test} RETURNING Widget__c (id WHERE (Name = 'New Widget') or (Id = '00Xz00000008Vq7' and Name = 'Widget Insert Test') or (Widget_Cost__c < 100 or Widget_Cost__c = null) ORDER BY Widget_Cost__c)`

```
FIND {test}
    RETURNING Widget__c (id WHERE createddate = THIS_FISCAL_QUARTER)
```

```
FIND {test}
    RETURNING Widget__c (id WHERE cf__c includes('AAA'))
```

```
FIND {test}
    RETURNING Widget__c (id), Model__c(Field1,Field2 WHERE Field1 = 'test' order by id
ASC, Name DESC)
```

```
FIND {test} IN ALL FIELDS
    RETURNING Widget__c(Name, Widget_Cost__c, id WHERE Name = 'test'),
        Model__c(Name, id),
        Part__c(id WHERE Part_Loc__c IN ('California', 'New York'))
```

```
FIND {test}
    RETURNING Widget__c (id WHERE (Name = 'New Widget')
        or (Id = '00Xz00000008Vq7'
        and Name = 'Widget Insert Test')
        or (Widget_Cost__c < 100 or Widget_Cost__c = null)
        ORDER BY Widget_Cost__c)
```

ORDER BY *clause*

You can 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`	Ascending (ASC) or descending (DESC) order of the results. Default order is ascending.

Syntax	Description
NULLS	Order null records at the beginning (first) or end (last) of the results. By default, NULL values are sorted last in ascending order and first in descending order.

Examples

The following example shows a single `ORDER BY` clause:

```
FIND {MyWidgetName} RETURNING Widget__c(Name, Id ORDER BY Id)
```

The following example shows multiple `ORDER BY` clauses:

```
FIND {MyModelName} RETURNING Model__c(Name, Id ORDER BY Name), Widget__c(Widget_Cost__c,
Id ORDER BY Widget_Cost__c)
```

The following search returns a result with widget records in alphabetical order by name, sorted in descending order, with widgets that have null names appearing last:

```
FIND {MyWidgetName} IN NAME FIELDS RETURNING Widget__c(Name, Id ORDER BY Name DESC NULLS
last)
```

UPDATE TRACKING

The `UPDATE TRACKING` clause is used to report on Salesforce Knowledge article searches and views. It allows developers to track the keywords used in the Salesforce Knowledge article search.

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 VIEWSTAT

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.

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
```

LIMIT *n*

The optional `LIMIT` clause allows you to specify the maximum number of rows returned in the text query, up to 2,000 results. If unspecified, then the default is 2,000 results , which 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. 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 widget records can be returned, and the remaining number of records can be models.

```
FIND {test} RETURNING Widget__c(id LIMIT 20), Model__c LIMIT 100
```

Object limits are evaluated in the order they are included in the query, and the maximum query limit is adjusted after each object is processed. For example, if only seven widgets match the query string above, then at most 93 model records can be returned. Likewise, if the following query returns 15 widgets and 30 models, then only 55 parts can be returned, regardless of the part object's limit of 75:

```
FIND {test} RETURNING Widget__c(id LIMIT 20), Model__c, Part__c(id LIMIT 75) LIMIT 100
```

If you specify a limit of 0, no records are returned for that object.

OFFSET *n*

Use the optional `OFFSET` to specify the starting row offset into the result set returned by your query. Using `OFFSET` is helpful for paging into large result sets, in scenarios where you need to quickly jump to a particular subset of the entire results. As 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 Widget__c(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 `MALFORMED_SEARCH: 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 Widget__c(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 Widget__c(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.

Querying Currency Fields in Multicurrency Organizations

If an organization is multicurrency enabled, you can use `convertCurrency()` in the *FieldList* of the `RETURNING` clause to convert currency fields to the user's currency.

Use this syntax for the `RETURNING` clause:

`convertCurrency(`***`Amount`***`)`

For example,

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

You cannot 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 organization:

```
WHERE Object_name Operator ISO_CODEvalue
```

For example:

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

In this example, part records will be returned if the record's currency `Amount` value is greater than the equivalent of USD5000. For example, a part with an amount of `USD5001` would be returned, but not `JPY7000`.

Use an ISO code that your organization has enabled and is active. If you do not put in an ISO code, then the numeric value is used instead of comparative amounts. Using the example above, part records with `JPY5001`, `EUR5001`, and `USD5001` would be returned. Note that if you use IN in a `WHERE` clause, you cannot mix ISO code and non-ISO code values.

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

Example Text Searches

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 `Widget Smith` in the name field of a widget, return the ID field of the records.

```
FIND {Widget Smith}
IN Name Fields
RETURNING Widget__c
```

Look for `Widget Smith` in the name field of a widget and return the name of any matching record that was also created in the current fiscal quarter.

```
FIND {Widget Smith}
IN Name Fields
RETURNING Widget__c (name Where createddate = THIS_FISCAL_QUARTER)
```

Look for `Widget Smith` or `Widget Smythe` in the name field of a widget or model and return the name. If a part record is called `Widget Smith` or `Widget Smythe`, the part record should not be returned.

```
FIND {"Widget Smith" OR "Widget Smythe"}
IN Name Fields
RETURNING Widget__c(name), Model__c(name)
```

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 in order to use them on the fly. Additionally, Apex script variables and expressions can be used if preceded by a colon (`:`).

Text Searches in CJK Languages

In Chinese, Japanese, and Korean (CJK), words are delimited by pairs of CJK-type characters.

Index