
Big Objects Implementation Guide

Version 64.0, Summer '25

Summer '25



CONTENTS

- Chapter 1: Big Objects** 1
 - Big Objects Best Practices 3
 - Define and Deploy Custom Big Objects 4
 - Deploying and Retrieving Metadata with the Zip File 10
 - Populate a Custom Big Object 12
 - Populate a Custom Big Object with Apex 12
 - Delete Data in a Custom Big Object 13
 - Big Objects Queueable Example 15
 - Big Object Query Examples 17
 - View Big Object Data in Reports and Dashboards 20
 - SOQL with Big Objects 21
- Chapter 2: API End-of-Life Policy** 23
- Index** 24

CHAPTER 1 Big Objects

In this chapter ...

- [Big Objects Best Practices](#)
- [Define and Deploy Custom Big Objects](#)
- [Populate a Custom Big Object](#)
- [Delete Data in a Custom Big Object](#)
- [Big Objects Queueable Example](#)
- [Big Object Query Examples](#)
- [View Big Object Data in Reports and Dashboards](#)
- [SOQL with Big Objects](#)

A big object stores and manages massive amounts of data on the Salesforce platform. You can archive data from other objects or bring massive datasets from outside systems into a big object to get a full view of your customers. Clients and external systems use a standard set of APIs to access big object data. A big object provides consistent performance, whether you have 1 million records, 100 million, or even 1 billion. This scale gives a big object its power and defines its features.

There are two types of big objects.

- Standard big objects—Objects defined by Salesforce and included in Salesforce products. `FieldHistoryArchive` is a standard big object that stores data as part of the Field Audit Trail product. Standard big objects are always available and can't be customized.
- Custom big objects—New objects that you create to store information unique to your org. Custom big objects extend the functionality that Lightning Platform provides. For example, if you're building an app to track product inventory, create a custom big object called `HistoricalInventoryLevels` to track historical inventory levels for analysis and future optimizations. This implementation guide is for configuring and deploying custom big objects.

EDITIONS

Available in: both Salesforce Classic and Lightning Experience

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

Custom Big Object Use Cases

- 360° view of the customer—Extend your Salesforce data model to include detailed information from loyalty programs, feeds, clicks, billing and provisioning information, and more.
- Auditing and tracking—Track and maintain a long-term view of Salesforce or product usage for analysis or compliance purposes.
- Historical archive—Maintain access to historical data for analysis or compliance purposes while optimizing the performance of your core CRM or Lightning Platform applications.

Differences Between Big Objects and Other Objects

Because a big object can store data on an unlimited scale, it has different characteristics than other objects, like sObjects. Big objects are also stored in a different part of the Lightning Platform.

Big Objects	sObjects
Horizontally scalable distributed database	Relational database
Non-transactional database	Transactional database
Hundreds of millions or even billions of records	Millions of records

These big object behaviors ensure a consistent and scalable experience.

- Big objects support only object and field permissions, not regular or standard sharing rules.
- Features like triggers, flows, processes, and the Salesforce mobile app aren't supported on big objects.
- When you insert an identical big object record with the same representation multiple times, only a single record is created so that writes can be idempotent. This behavior is different from an sObject, which creates a record for each request to create an object.

API Support for Big Objects

It's easy to integrate custom big objects with your live Salesforce data. You can process big objects with SOQL, Bulk, Chatter and SOAP APIs.



Note: These APIs are the only APIs supported for big objects. The REST API, for example, isn't supported.

SEE ALSO:

[Release Notes: Field History Tracking Data Deleted After 18 Months](#)

[CustomObject](#)

[Salesforce Help: Big Object Support in Analytics](#)

Big Objects Best Practices

A big object is unique because of its ability to scale for massive amounts of data.

Considerations When Using Big Objects

- To define a big object or add a field to a custom big object, use either Metadata API or Setup.
- Big objects support custom Lightning and Visualforce components rather than standard UI elements home pages, detail pages, or list views.
- You can create up to 100 big objects per org. The limits for big object fields are similar to the limits on custom objects and depend on your org's license type.
- You can't use Salesforce Connect external objects to access big objects in another org.
- Big objects don't support encryption. If you archive encrypted data from a standard or custom object, it's stored as clear text on the big object.

If you're using Salesforce Shield Platform Encryption, standard or custom object field history is encrypted. For field history, data is archived using the Shield field history archive. Big objects respect encryption at rest.

Shield Platform Encryption isn't otherwise supported for custom big objects.

EDITIONS

Available in: both Salesforce Classic and Lightning Experience



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

Design with Resiliency in Mind

The big objects database stores billions of records and is a distributed system that favors consistency over availability. The database is designed to ensure row-level consistency.

When working with big data and writing batches of records using APIs or Apex, you can experience a partial batch failure while some records are written and others aren't. Because the database is highly responsive and consistent at scale, this type of behavior is expected. In these cases, simply retry until all records are written.

Keep these principles in mind when working with big objects.

- The best practice when writing to a big object is to have a retry mechanism in place. Retry the batch until you get a successful result from the API or Apex method.
-  **Tip:** To add logging to a custom object and surface errors to users, use the `addError()` method. See [An Introduction to Exception Handling](#).
-  **Tip:** To verify that all records are saved, check the `Database.SaveResult` class. See [SaveResult Class Reference](#).
- Don't try to figure out which records succeeded and which failed. Retry the entire batch.
 - Big objects don't support transactions. If attempting to read or write to a big object using a trigger, process, or flow on a sObject, use asynchronous Apex. Asynchronous Apex has features like the `Queueable` interface that isolates DML operations on different sObject types to prevent the mixed DML error.
 - Because your client code must retry, use asynchronous Apex to write to a big object. By writing asynchronously, you're better equipped to handle database lifecycle events.

SEE ALSO:

[Salesforce Help: Big Objects](#)

Define and Deploy Custom Big Objects

You can define custom big objects with Metadata API or in Setup. After you define and deploy a big object, you can view it or add fields in Setup. After you've deployed a big object, you can't edit or delete the index. To change the index, start over with a new big object. To define a big object in Setup, see Salesforce Help.

EDITIONS

Available in: both Salesforce Classic and Lightning Experience

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

Define a Custom Big Object

Define a custom big object through Metadata API by creating XML files that contain its definition, fields, and index.

- **object files**—Create a file for each object to define the custom big object, its fields, and its index.
- **permissionset/profile files**—Create a permission set or profile file to specify permissions for each field. These files aren't required, but they're required to grant access to users. By default, access to a custom big object is restricted.
- **package file**—Create a file for Metadata API to specify the contents of the metadata you want to migrate.



Note: The package file is unrelated to the packaging feature for Salesforce. This file isn't an unlocked, unmanaged, or managed package. It's simply a file used by Metadata API.



Note: While custom big objects use the CustomObject metadata type, some parameters are unique to big objects and others aren't applicable. The specific metadata parameters that apply to big objects are outlined in this document.



Naming Conventions for Custom Big Objects


Object names must be unique across all standard objects, custom objects, external objects, and big objects in the org. In the API, the names of custom big objects have a suffix of two underscores immediately followed by a lowercase "b" (__b). For example, a big object named "HistoricalInventoryLevels" is seen as HistoricalInventoryLevels__b in that organization's WSDL. We recommend that you make object labels unique across all objects in the org - standard, custom, external and big objects.

CustomObject Metadata

Field Name	Field Type	Description
deploymentStatus	DeploymentStatus (enumeration of type string)	Custom big object's deployment status (Deployed for all big objects)
fields	CustomField[]	Definition of a field in the big object
fullName	string	Unique API name of the big object
indexes	Index[]	Definition of the index
label	string	Big object's name as displayed in the UI
pluralLabel	string	Field plural name as displayed in the UI

CustomField Metadata

Field Name	Field Type	Description
fullName	string	Unique API name of a field.
label	string	Field name as displayed in the UI.
length	int	Length of a field in characters (Text and LongTextArea fields only). The total number of characters across all text fields in an index can't exceed 100. To increase this value, contact Salesforce Customer Support.  Note: Email fields are 80 characters. Phone fields are 40 characters. Keep these lengths in mind when designing your index because they count toward the 100 character limit.
pluralLabel	string	Field plural name as displayed in the UI.
precision	int	Number of digits for a number value. For example, the number 256.99 has a precision of 5 (number fields only).
referenceTo	string	Related object type for a lookup field (lookup fields only).
relationshipName	string	Name of a relationship as displayed in the UI (lookup fields only).
required	boolean	Specifies whether the field is required. All fields that are part of the index must be marked as required.
scale	int	Number of digits to the right of the decimal point for a number value. For example, the number 256.99 has a scale of 2 (number fields only).
type	FieldType	Field type. Supports DateTime, Email, Lookup, Number, Phone, Text, LongTextArea, and URL.  Note: You can't include LongTextArea and URL fields in the index.

 **Note:** Uniqueness isn't supported for custom fields.

Index Metadata


Represents an index defined within a custom [big object](#). Use this metadata type to define the composite primary key (index) for a custom big object.

Field Name	Field Type	Description
fields	IndexField[]	The definition of the fields in the index.
label	string	Required. This name is used to refer to the big object in the user interface. Available in API version 41.0 and later.

IndexField Metadata

Defines which fields make up the index, their order, and sort direction. The order in which the fields are defined determines the order fields are listed in the index.

 **Note:** The total number of characters across all text fields in an index can't exceed 100. To increase this value, contact Salesforce Customer Support.

Field Name	Field Type	Description
name	string	Required. The API name for the field that's part of the index. This value must match the <code>fullName</code> value for the corresponding field in the fields section and be marked as required.  Warning: When querying a big object record via SOQL and passing the results as arguments to the delete API, if any index field name has a leading or trailing white space, you can't delete the big object record.
sortDirection	string	Required. The sort direction of the field in the index. Valid values are <code>ASC</code> for ascending order and <code>DESC</code> for descending order.

Example: Create Metadata Files for Deployment

The following XML excerpts create metadata files that you can deploy. Each Customer Interaction object represents customer data from a single session in an online video game. The `Account__c`, `Game_Platform__c`, and `Play_Date__c` fields define the index, and a lookup field relates the Customer Interactions to the Account object.

Customer_Interaction__b.object

```
<?xml version="1.0" encoding="UTF-8"?>
<CustomObject xmlns="http://soap.sforce.com/2006/04/metadata">
  <deploymentStatus>Deployed</deploymentStatus>

  <fields>
    <fullName>In_Game_Purchase__c</fullName>
    <label>In-Game Purchase</label>
    <length>16</length>
    <required>false</required>
    <type>Text</type>
    <unique>false</unique>
  </fields>

  <fields>
    <fullName>Level_Achieved__c</fullName>
    <label>Level Achieved</label>
    <length>16</length>
    <required>false</required>
    <type>Text</type>
    <unique>false</unique>
  </fields>

  <fields>
    <fullName>Lives_This_Game__c</fullName>
```

```

    <label>Lives Used This Game</label>
    <length>16</length>
    <required>false</required>
    <type>Text</type>
    <unique>false</unique>
</fields>

<fields>
    <fullName>Game_Platform__c</fullName>
    <label>Platform</label>
    <length>16</length>
    <required>true</required>
    <type>Text</type>
    <unique>false</unique>
</fields>

<fields>
    <fullName>Score_This_Game__c</fullName>
    <label>Score This Game</label>
    <length>16</length>
    <required>false</required>
    <type>Text</type>
    <unique>false</unique>
</fields>

<fields>
    <fullName>Account__c</fullName>
    <label>User Account</label>
    <referenceTo>Account</referenceTo>
    <relationshipName>Game_User_Account</relationshipName>
    <required>true</required>
    <type>Lookup</type>
</fields>

<fields>
    <fullName>Play_Date__c</fullName>
    <label>Date of Play</label>
    <required>true</required>
    <type>DateTime</type>
</fields>

<fields>
    <fullName>Play_Duration__c</fullName>
    <label>Play Duration</label>
    <required>false</required>
    <type>Number</type>
    <scale>2</scale>
    <precision>18</precision>
</fields>

<indexes>
    <fullName>CustomerInteractionsIndex</fullName>
    <label>Customer Interactions Index</label>
    <fields>

```

```

        <name>Account__c</name>
        <sortDirection>DESC</sortDirection>
    </fields>
    <fields>
        <name>Game_Platform__c</name>
        <sortDirection>ASC</sortDirection>
    </fields>
    <fields>
        <name>Play_Date__c</name>
        <sortDirection>DESC</sortDirection>
    </fields>
</indexes>

<label>Customer Interaction</label>
<pluralLabel>Customer Interactions</pluralLabel>
</CustomObject>

```

package.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<Package xmlns="http://soap.sforce.com/2006/04/metadata">
    <types>
        <members>*</members>
        <name>CustomObject</name>
    </types>
    <types>
        <members>*</members>
        <name>PermissionSet</name>
    </types>
    <version>41.0</version>
</Package>

```

Customer_Interaction_BigObject.permissionset

```

<?xml version="1.0" encoding="UTF-8"?>
<PermissionSet xmlns="http://soap.sforce.com/2006/04/metadata">

    <label>Customer Interaction Permission Set</label>

    <fieldPermissions>
        <editable>true</editable>
        <field>Customer_Interaction__b.In_Game_Purchase__c</field>
        <readable>true</readable>
    </fieldPermissions>

    <fieldPermissions>
        <editable>true</editable>
        <field>Customer_Interaction__b.Level_Achieved__c</field>
        <readable>true</readable>
    </fieldPermissions>

    <fieldPermissions>
        <editable>true</editable>
        <field>Customer_Interaction__b.Lives_This_Game__c</field>
        <readable>true</readable>
    </fieldPermissions>

```

```

</fieldPermissions>

<fieldPermissions>
  <editable>true</editable>
  <field>Customer_Interaction__b.Play_Duration__c</field>
  <readable>true</readable>
</fieldPermissions>

<fieldPermissions>
  <editable>true</editable>
  <field>Customer_Interaction__b.Score_This_Game__c</field>
  <readable>true</readable>
</fieldPermissions>

</PermissionSet>

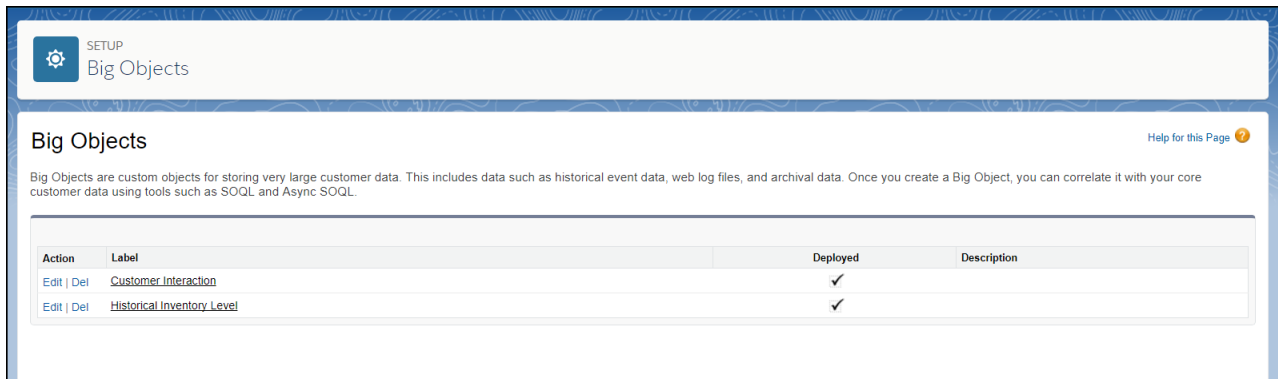
```

Deploy Custom Big Objects Using Metadata API

Use Metadata API and the [Ant Migration Tool](#) to deploy. When building files to deploy a custom big object, make sure the `object` file is in a folder called `objects` and the `permissionset` file is in a folder called `permissionsets`. Put the `package.xml` file in the root directory and not in a subfolder.

View a Custom Big Object in Setup

After you've deployed your custom big object, you can view it by logging in to your organization and, from Setup, entering *Big Objects* in the Quick Find box, then selecting **Big Objects**.



SETUP
Big Objects

Big Objects [Help for this Page](#)

Big Objects are custom objects for storing very large customer data. This includes data such as historical event data, web log files, and archival data. Once you create a Big Object, you can correlate it with your core customer data using tools such as SOQL and Async SOQL.

Action	Label	Deployed	Description
Edit Del	Customer Interaction	✓	
Edit Del	Historical Inventory Level	✓	

To see its fields and relationships, click the name of a big object.

Big Object Definition Detail

Singular Label	Customer Interaction	Description	
Plural Label	Customer Interactions	Deployment Status	Deployed
Object Name	Customer_Interaction		
API Name	Customer_Interaction__b		
Created By	Admin User, 9/11/2017 1:37 PM	Modified By	Admin User, 9/11/2017 1:38 PM

Standard Fields

No standard fields defined

Custom Fields & Relationships

Action	Field Label	API Name	Data Type	Indexed	Index Position	Index Direction	Modified By
Edit	Date of Play	Play_Date__c	Date/Time	✓	3	DESC	Admin User, 9/11/2017 1:37 PM
Edit	In-Game Purchase	In_Game_Purchase__c	Text(16)				Admin User, 9/11/2017 1:37 PM
Edit	Level Achieved	Level_Achieved__c	Text(16)				Admin User, 9/11/2017 1:37 PM
Edit	Lives Used This Game	Lives_This_Game__c	Text(16)				Admin User, 9/11/2017 1:37 PM
Edit	Platform	Game_Platform__c	Text(16)	✓	2	ASC	Admin User, 9/11/2017 1:37 PM
Edit	Play Duration	Play_Duration__c	Number(16, 2)				Admin User, 9/11/2017 1:37 PM
Edit	Score This Game	Score_This_Game__c	Text(16)				Admin User, 9/11/2017 1:37 PM
Edit	User Account	Account__c	Lookup(Account)	✓	1	DESC	Admin User, 9/11/2017 1:37 PM

SEE ALSO:

[CustomObject](#)

[PermissionSet](#)

[Index](#)

[Deploying and Retrieving Metadata with the Zip File](#)

[Salesforce Help: Big Objects](#)

Deploying and Retrieving Metadata with the Zip File

The `deploy()` and `retrieve()` calls are used to deploy and retrieve a .zip file. Within the .zip file is a project manifest (`package.xml`) that lists what to retrieve or deploy, and one or more XML components that are organized into folders.

Note: A component is an instance of a metadata type. For example, `CustomObject` is a metadata type for custom objects, and the `MyCustomObject__c` component is an instance of a custom object.



The files that are retrieved or deployed in a .zip file might be unpackaged components that reside in your org (such as *standard objects*) or packaged components that reside within named packages.

Note: You can deploy or retrieve up to 10,000 files at once. Managed packages use different limits: First-generation managed packages that have passed AppExchange Security Review can contain up to 35,000 files. Second-generation managed packages can contain up to 10,000 files. The maximum size of the deployed or retrieved .zip file is 39 MB. If the files are uncompressed in an unzipped folder, the size limit is 600 MB.

- If using the Ant Migration Tool to deploy an unzipped folder, all files in the folder are compressed first. The maximum size of uncompressed components in an unzipped folder is 600 MB or less depending on the compression ratio. If the files have a high compression ratio, you can migrate a total of approximately 600 MB because the compressed size would be under 39 MB. However, if the components can't be compressed much, like binary static resources, you can migrate less than 600 MB.

- Metadata API base-64 encodes components after they're compressed. The resulting .zip file can't exceed 50 MB, which is the limit for SOAP messages. Base-64 encoding increases the size of the payload, so your compressed payload can't exceed approximately 39 MB before encoding.
- You can perform a `retrieve()` call for a big object only if its index is defined. If a big object is created in Setup and doesn't yet have an index defined, you can't retrieve it.
- Limits can change without notice.


Every .zip file contains a project manifest, a file that's named `package.xml`, and a set of directories that contain the components. The manifest file defines the components that you're trying to retrieve or deploy in the .zip file. The manifest also defines the API version that's used for the deployment or retrieval.

-  **Note:** You can edit the project manifest, but be careful if you modify the list of components it contains. When you deploy or retrieve components, Metadata API references the components listed in the manifest, not the directories in the .zip file.
-  **Note:** Note: If you're retrieving any components that have dependencies by using the `rootTypesWithDependencies` parameter in the `RetrieveRequest` object, the dependent metadata components are added to the returned .zip file and `package.xml` file in the same directory as the root type that's being retrieved. This directory has a JSON file for each component with dependencies in the format `ComponentName.roottype.dependencies-meta.json`.

The following is a sample `package.xml` file. You can retrieve an individual component for a metadata type by specifying its `fullName` field value in a `members` element. You can also retrieve all components of a metadata type by using `<members>*</members>`.

```
<?xml version="1.0" encoding="UTF-8"?>
<Package xmlns="http://soap.sforce.com/2006/04/metadata">
  <types>
    <members>MyCustomObject__c</members>
    <name>CustomObject</name>
  </types>
  <types>
    <members>*</members>
    <name>CustomTab</name>
  </types>
  <types>
    <members>Standard</members>
    <name>Profile</name>
  </types>
  <version>64.0</version>
</Package>
```

The following elements can be defined in `package.xml`.

- `<fullName>` contains the name of the server-side package. If no `<fullName>` exists, the `package.xml` defines a client-side unpackaged package.
 - `<types>` contains the name of the metadata type (for example, `CustomObject`) and the named members (for example, `myCustomObject__c`) to be retrieved or deployed. You can add multiple `<types>` elements in a manifest file.
 - `<members>` contains the `fullName` of the component, for example `MyCustomObject__c`. The `listMetadata()` call is useful for determining the `fullName` for components of a particular metadata type if you want to retrieve an individual component. For many metadata types, you can replace the value in `members` with the wildcard character `*` (asterisk) instead of listing each member separately. See the reference topic for a specific type to determine whether that type supports wildcards.
-  **Note:** You specify Security in the `<members>` element and Settings in the name element when retrieving the `SecuritySettings` component type.

- `<name>` contains the metadata type, for example `CustomObject` or `Profile`. There is one name defined for each metadata type in the directory. Any metadata type that extends `Metadata` is a valid value. The name that's entered must match a metadata type that's defined in the Metadata API WSDL. See [Metadata Types](#) for a list.
- `<version>` is the API version number that's used when the .zip file is deployed or retrieved. Currently the valid value is `64.0`.


For more sample `package.xml` manifest files that show you how to work with different subsets of metadata, see [Sample package.xml Manifest Files](#).

To delete components, see [Deleting Components from an Organization](#).

Populate a Custom Big Object

Use Salesforce APIs to populate a custom big object.

You can use a CSV file to load data into a custom big object via Bulk API 2.0. The first row in the CSV file must contain the field labels used to map the CSV data to the fields in the custom big object during import.

 **Note:** Both Bulk API and Bulk API 2.0 support querying and inserting big objects.

Insertion is idempotent, so inserting data that already exists won't result in duplicates. Reinserting is helpful when uploading millions of records. If an error occurs, the reinsert reuploads the failed uploads without duplicate data. During the reinsertion, if no record exists for the provided index, a new record is inserted.


For example, this CSV file contains data for import into a Customer Interaction big object.

```
Play Start,In-Game Purchase,Level Achieved,Lives Used,Platform,Play Stop,Score,Account
2015-01-01T23:01:01Z,A12569,57,7,PC,2015-01-02T02:27:01Z,55736,001R000000302D3
2015-01-03T13:22:01Z,B78945,58,7,PC,2015-01-03T15:47:01Z,61209,001R000000302D3
2015-01-04T15:16:01Z,D12156,43,5,iOS,2015-01-04T16:55:01Z,36148,001R000000302D3
```

Populate a Custom Big Object with Apex

Use Apex to populate a custom big object.

You can create and update custom big object records in Apex using the `Database.insertImmediate()` method.

 **Warning:** Apex tests that use mixed DML calls aren't allowed and fail. If you write only to the Big Object, the test inserts bad data into the target big object that you then must delete manually. To contain test DML calls to the target big object, use a mocking framework with the stub API instead.

We recommend that you use the `Apex String.trim()` method to remove leading and trailing white space before you insert values, especially for values in primary key fields. This best practice ensures that SOQL queries on the big objects later work as you expect.

When you specify an index field in a SOQL query WHERE clause, SOQL removes any leading or trailing white space before it compares it to the actual field value. Even if your filter string matches the value that was inserted, leading and trailing white space isn't compared, and so no rows are matched by the filter.

Therefore, leading and trailing white space can't be compared to values that have been stored. Even if your filter string matches the value that was inserted, no rows are matched by the filter.

If you set values to NULL when upserting into a custom big object, the fields aren't updated if they have existing values. To set these values to NULL, delete the field and recreate it.

Reinserting a record with the same index but different data results in behavior similar to an upsert operation. If a record with the index exists, the insert overwrites the index values with the new data. Insertion is idempotent, so inserting data that exists doesn't result in

duplicates. Reinserting is helpful when uploading millions of records. If an error occurs, the reinsertion reuploads the failed uploads without duplicate data. During the reinsertion, if no record exists for the provided index, a new record is inserted.

If a record insert fails, the `Database.insertImmediate()` method doesn't throw an exception. Instead, it returns a `SaveResult` object that has a `getErrors()` method that returns a list of `Database.Error` objects. Each `Database.Error` object contains information about an error that occurred as a result of a failed record insert. See the [Apex developer guide](#) for more information about `SaveResult` and an example of iterating through the errors.

Here's an example of an insert operation in Apex that assumes a table where the index consists of `FirstName__c`, `LastName__c`, and `Address__c`.

```
// Define the record.
PhoneBook__b pb = new PhoneBook__b();
pb.FirstName__c = 'John';
pb.LastName__c = 'Smith';
pb.Address__c = '1 Market St';
pb.PhoneNumber__c = '555-1212';
database.insertImmediate(pb);
// A single record will be created in the big object.
```

```
// Define the record with the same index values but different phone number.
PhoneBook__b pb = new PhoneBook__b();
pb.FirstName__c = 'John';
pb.LastName__c = 'Smith';
pb.Address__c = '1 Market St';
pb.PhoneNumber__c = '415-555-1212';
database.insertImmediate(pb);
// The existing records will be "re-inserted". Only a single record will remain in the big object.
```

```
// Define the record with the different index values and different phone number
PhoneBook__b pb = new PhoneBook__b();
pb.FirstName__c = 'John';
pb.LastName__c = 'Smith';
pb.Address__c = 'Salesforce Tower';
pb.PhoneNumber__c = '415-555-1212';
database.insertImmediate(pb);
// A new record will be created leaving two records in the big object.
```

SEE ALSO:

[Build a Mocking Framework with the Stub API](#)


[Apex Developer Guide: Returned Database Errors](#)

Delete Data in a Custom Big Object

Use Apex or SOAP to delete data in a custom big object.

The Apex method `deleteImmediate()` deletes data in a custom big object. Declare an `sObject` that contains all the fields in the custom big object's index. The `sObject` acts like a template. All rows that match the `sObject`'s fields and values are deleted. You can specify only fields that are part of the big object's index. You must specify all fields in the index. You can't include a partially specified index or non-indexed field, and wildcards aren't supported.

If you're deleting all records because of capacity optimization, insert one or two blank records after deletion and wait 24 hours for the new capacity to be recognized.

 **Important:** The batch limit for big objects using `deleteImmediate()` is 50,000 records at a time.

 **Note:** These examples assume that when you initially inserted big object values you used the `Apex String.trim()` method to remove leading and trailing white space in the index fields. See [Populate a Custom Big Object with Apex](#) on page 12.

In this example, `Account__c`, `Game_Platform__c`, and `Play_Date__c` are part of the custom big object's index. When specifying specific values after the `WHERE` clause, fields must be listed in the order they appear in the index, without any gaps.


```
// Declare sObject using the index of the custom big object -->
List<Customer_Interaction__b> cBO = new List<Customer_Interaction__b>();
cBO.addAll([SELECT Account__c, Game_Platform__c, Play_Date__c FROM Customer_Interaction__b
WHERE Account__c = '001d000000Ky3xIAB']);

Database.deleteImmediate(cBO);
```

To use the SOAP call `deleteByExample()`, declare an `sObject` that contains the fields and values to delete. The `sObject` acts like a template. All rows that match the `sObject`'s fields and values are deleted. You can specify only fields that are part of the big object's index. All fields in the index must be specified. You can't include a partially specified index or non-indexed field, and wildcards aren't supported. This example deletes all rows in which `Account__c` is 001d000000Ky3xIAB, `Game_Platform__c` is iOS, and `Play_Date__c` is 2017-11-28T19:13:36.000z.

Java example code:

```
public static void main(String[] args) {
    try{
        Customer_Interaction__b[] sObjectsToDelete = new Customer_Interaction__b[1];
        //Declare an sObject that has the values to delete
        Customer_Interaction__b customerBO = new Customer_Interaction__b();
        customerBO.setAccount__c ("001d000000Ky3xIAB");
        customerBO.setGame_Platform__c ("iOS");
        Calendar dt = new GregorianCalendar(2017, 11, 28, 19, 13, 36);
        customerBO.setPlay_Date__c(dt);
        sObjectsToDelete[0] = customerBO;
        DeleteByExampleResult[] result = connection.deleteByExample(sObjectsToDelete);
    } catch (ConnectionException ce) {
        ce.printStackTrace();
    }
}
```


 **Note:** Repeating a successful `deleteByExample()` operation produces a success result, even if the rows were already deleted.


SEE ALSO:

https://developer.salesforce.com/docs/atlas.en-us.api.meta/api/sforce_api_calls_deletebyexample.htm

Big Objects Queueable Example

To read or write to a big object using a trigger, process, or flow from a sObject, use asynchronous Apex. This example uses the asynchronous Apex `Queueable` interface to isolate DML operations on different sObject types to prevent the mixed DML error.

 **Example:** This trigger occurs when a case record is inserted. It calls a method to insert a batch of big object records and demonstrates a partial failure case in which some records succeed and some fail. To create metadata files for the `Customer_Interaction__b` object in this example, use the XML excerpts in the [Create Metadata Files for Deployment](#) on page 6 example.

 **Tip:** To add logging to a custom object and surface errors to users, use the `addError()` method. See [An Introduction to Exception Handling](#).

```
// CaseTrigger.apxt

trigger CaseTrigger on Case (before insert) {
    if (Trigger.operationType ==
        TriggerOperation.BEFORE_INSERT) {
        // Customer_Interaction__b has three required fields
        in its row key, in this order:
        // 1) Account__c - lookup to Account
        // 2) Game_Platform__c - Text(18)
        // 3) Play_Date__c - Date/Time
        List<Customer_Interaction__b> interactions = new
        List<Customer_Interaction__b>();

        // Assemble the list of big object records to be
        inserted
        for (Case c : Trigger.new) {
            Customer_Interaction__b ci = new
            Customer_Interaction__b(
                Account__c = c.AccountId,
                // In this example, the Case object has a
                custom field, also named Game_Platform__c
                Game_Platform__c = c.Game_Platform__c,
                Play_Date__c = Date.today()
            );
            interactions.add(ci);
        }

        // CustomerInteractionHandler is an asynchronous
        queueable Apex class
        CustomerInteractionHandler handler = new
        CustomerInteractionHandler(interactions);
        System.enqueueJob(handler);
    }
}
```

EDITIONS

Available in: both Salesforce Classic and Lightning Experience

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

The trigger uses the `Queueable` Apex interface to asynchronously call a method to insert into a big object.

```
// CustomerInteractionHandler.apxc

public class CustomerInteractionHandler implements Queueable {

    private List<Customer_Interaction__b> interactions;

    public CustomerInteractionHandler(List<Customer_Interaction__b> interactions) {
        this.interactions = interactions;
    }

    /*
     * Here we insert the Customer Interaction big object records,
     * or log an error if insertion fails.
     */
    public void execute(QueueableContext context){

        List<ExceptionStorage__c> errors = new List<ExceptionStorage__c>();

        try {
            // We have to use insertImmediate() to insert big object records.
            List<Database.SaveResult> srList = Database.insertImmediate(interactions);

            // Check the save results from the bulk insert
            for (Database.SaveResult sr: srList) {
                if (sr.isSuccess()) {
                    System.debug('Successfully inserted Customer Interaction.');
```

```
                } else {
                    for (Database.Error err : sr.getErrors()) {
                        // Display an error message if the insert failed
                        System.debug(err.getStatusCode() + ': ' + err.getMessage() +
'; ' +
                                'Error fields: ' + err.getFields());

                        // Write to a custom object, such as ExceptionStorage__c
                        // for a more durable record of the failure
                        ExceptionStorage__c es = new ExceptionStorage__c(
                            name = 'Error',
                            ExceptionMessage__c = (err.getMessage()).abbreviate(255),

                            ExceptionType__c = String.valueOf(err.getStatusCode()),

                            ExceptionFields__c =
                                (String.valueOf(err.getFields())) .abbreviate(255)
                        );
                        errors.add(es);
                    }
                }
            }
        } catch (Exception e) {
            // Exception occurred, output the exception message
            System.debug('Exception: ' + e.getTypeName() + ', ' + e.getMessage());
        }
    }
}
```

```

        // Write any errors to a custom object as well
        ExceptionStorage__c es = new ExceptionStorage__c(
            name = 'Exception',
            ExceptionMessage__c = e.getMessage(),
            ExceptionType__c = e.getTypeName()
        );
        errors.add(es);
    }

    // If any errors occurred, save the ExceptionStorage records
    if (errors.size() > 0) {
        insert errors;
    }
}

```

SEE ALSO:

[Apex Developer Guide: Queueable Apex](#)

Big Object Query Examples

Understand some of the common big object querying use cases.

Customer 360 Degree and Filtering

In this use case, administrators load various customer engagement data from external sources into Salesforce big objects and then process the data to enrich customer profiles in Salesforce. The goal is to store customer transactions and interactions—such as point-of-sale data, orders, and line items—in big objects, and then process and correlate that data with your core CRM data. Anchoring customer transactions and interactions with core CRM data provides a richer 360-degree view that translates to an enhanced customer experience.

Batch Apex is the best choice for automated processing on a big object or `ApiEvent`, `ReportEvent`, or `ListViewEvent`. This example shows how to add processing that references correlated data.

Run a batch Apex query on a big object and correlate the Contact information associated with that big object.

```

public class QueryBigObjectAndContact implements Database.Batchable<SObject> {
    private String key;

    public QueryBigObjectAndContact(String keyParam) {
        key = keyParam
    }

    public Iterable<SObject> start(Database.BatchableContext BC) {
        return [SELECT Big_Object_Field__c, Account__c FROM Big_Object__b WHERE
Big_Object_Primary_Key > key LIMIT 50000]
    }

    public void execute(Database.BatchableContext bc, List<Big_Object__b> bos){
        // process the batch of big objects and associate them to Accounts
        Map<Id, Big_Object__b> accountIdToBigObjectMap = new Map<Id, Big_Object__b>();
        for (Big_Object__b bigObject : bos) {

```

```

        accountIdToBigObjectMap.put(bigObject.Account__c, bigObject);
        key = bigObject.Big_Object_Primary_Key__c
    }
    Map<Id, Account> accountMap = new Map<Id, Account>(
        [SELECT Id, Name, ... FROM Account WHERE Id IN :accountIdToBigObjectMap.keySet()]

    );
    for (Id accountId : accountMap.keySet()) {
        Big_Object__b bigObject = accountIdToBigObjectMap.get(accountId);
        Account account = accountMap.get(accountId);
        // perform any actions that integrate the big object and Account
    }
}
public void finish(Database.BatchableContext bc){
    // You can daisy chain additional calls using the primary key of the big object
    // to get around the 50k governor limit
    QueryBigObjectAndContact nextBatch = new QueryBigObjectAndContact(key);
    Database.executeBatch(nextBatch);
}
}

```

Field Audit Trail

This example shows how to query `FieldHistoryArchive` and analyze a large number of results in a CSV format.

Example URI

```
/services/data/vXX.X/jobs/query
```

Example Post Request

```

{
  "operation": "query",
  "query": "SELECT ParentId, FieldHistoryType, Field, Id, NewValue, OldValue FROM
FieldHistoryArchive WHERE FieldHistoryType = 'Account' AND CreatedDate > LAST_MONTH"
}

```

Use the [Get Results for a Query Job](#) resource.

Example CURL Request

```

curl --include --request GET \
--header "Authorization: Bearer token" \
--header "Accept: text/csv" \
https://instance.salesforce.com/services/data/vXX.X/jobs/query/750R0000000zxr8IAA/results
?maxRecords=50000

```

This request results in a CSV file that can be examined for auditing purposes.

Real-Time Event Monitoring

With Real-Time Event Monitoring you can track who is accessing confidential and sensitive data in your Salesforce org. You can view information about individual events or track trends in events to swiftly identify unusual behavior and safeguard your company's data. These features are useful for compliance with regulatory and audit requirements.

With Real-Time Events, you can monitor data accessed through API calls, report executions, and list views. The corresponding event objects are called `ApiEvent`, `ReportEvent`, and `ListViewEvent`. Querying these events covers many common scenarios because more than 50% of SOQL queries occur using the SOAP, REST, or Bulk APIs. Key information about each query—such as the username, user ID, rows processed, queried entities, and source IP address—is stored in the event objects. You can then run SOQL queries on the event objects to discover user activity details.

For more information, see [Real-Time Event Monitoring](#).

This example shows how to query and analyze an event big object using a field's contents.

```
public class EventMatchesObject implements Database.Batchable<sObject> {
    private String lastEventDate;

    public EventMatchesObject(String lastEventDateParam) {
        lastEventDate = lastEventDateParam;
    }

    public Iterable<sObject> start(Database.BatchableContext bc) {
        return [SELECT EventDate, EventIdentifier, QueriedEntities, SourceIp, Username,
        UserAgent FROM ApiEvent WHERE EventDate > lastEventDate LIMIT 50000]
    }

    public void execute(Database.BatchableContext bc, List<ApiEvent> events){
        // Process this list of entities if a certain attribute matches
        for (ApiEvent event: events) {
            String objectString = 'Patent__c';
            String eventIdentifier = event.EventIdentifier;
            if (eventIdentifier.contains(objectString) {
                // Perform actions on the event that contains 'Patent__c'
            }
            lastEventDate = format(event.EventDate);
        }
    }

    public void finish(Database.BatchableContext bc){
        // You can daisy chain additional calls using EventDate or other filter fields to
        get around the 50k governor limit
        EventMatchesObject nextBatch = new EventMatchesObject(lastEventDate);
        Database.executeBatch(nextBatch);
    }
}
```

Aggregate Queries

This example shows an alternative for aggregate queries similar to the `COUNT()` method.

```
public class CountBigObjects implements Database.Batchable<sObject> {
    private Integer recordsCounted;
    private String key;

    public CountBigObjects(Integer recordsCountedParam, String keyParam) {
        recordsCounted = recordsCountedParam
        key = keyParam
    }
}
```

```

public Iterable<SObject> start(Database.BatchableContext bc) {
    return [SELECT Custom_Field__c FROM Big_Object__b LIMIT 25000]
}

public void execute(Database.BatchableContext bc, List<Big_Object__b> bos){
    // process the batch of big objects and associate them to Accounts
    Map<Id, Big_Object__b> accountIdToBigObjectMap = new Map<Id, Big_Object__b>();
    for (Big_Object__b bigObject : bos) {
        accountIdToBigObjectMap.put(bigObject.Account__c, bigObject);
    }
    Map<Id, Account> accountMap = new Map<Id, Account>(
        [SELECT Id, Name, ... FROM Account WHERE Id IN :accountIdToBigObjectMap.keySet()]
    );
    for (Id accountId : accountMap.keySet()) {
        Big_Object__b bigObject = accountIdToBigObjectMap.get(accountId);
        Account account = accountMap.get(accountId);
        // perform any actions that integrate the big object and Account
    }
}

public void finish(Database.BatchableContext bc) {
    // You can daisy chain additional calls using the primary key of the big object
    // to get around the 50k governor limit
    CountBigObjects nextBatch = new CountBigObjects(recordsCounted, key);
    Database.executeBatch(nextBatch);
}
}

```

View Big Object Data in Reports and Dashboards

When working with big data and billions of records, it's not practical to build reports or dashboards directly from that data. Instead, use Bulk API to write a query that extracts a smaller, representative subset of the data that you're interested in. You can store this working dataset in a custom object and use it in reports, dashboards, or any other Lightning Platform feature.

1. Identify the big object that contains the data for which you need a report. In this example, the `Ride__b` big object contains the full dataset.
2. Create a custom object. This object holds the working dataset for the big object data that you want to report on. In this example, we use the `Bike_Rental__c` custom object.
 - a. Under Optional Features for the custom object, click **Allow Reports**.
 - b. Add custom fields to the object that match the fields that you want to report on from the big object.
3. Create an SOQL query that builds your working dataset by pulling the data from your big object into your custom object.



Tip: To ensure that your working dataset is always up-to-date for accurate reporting, set this job to run nightly.

4. Build a report using the working dataset you created.
 - a. From Setup, enter *Report Types* in the Quick Find box, then select **Report Types**.
 - b. Create a custom report type.
 - c. For the Primary Object, select the custom object from step 2, `Bike_Rental__c`.

- d. Set the report to **Deployed**.
- e. Run the report.

You can now use the information from your working dataset not only in your reports, but also in dashboards or any other Lightning Platform feature.

SOQL with Big Objects

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

Build an index query, starting from the first field defined in the index, without gaps between the first and last field in the query. You can use `=` or `IN` on any field in your query, although you can use `IN` only one time. You can use the range operations `<`, `>`, `<=`, or `>=` only on the last field of your query.



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

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

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

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

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

These queries assume that you have a table where the index is defined by `LastName__c`, `FirstName__c`, and `PhoneNumber__c`.

This query specifies all three fields in the index. In this case, the filter on `PhoneNumber__c` can use a range operator.

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

This query specifies only the first two fields in the index. In this case, the filter on `FirstName__c` can use a range operator.

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

This query specifies only the first field in the index. The filter on `LastName__c` can use a range operator.

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

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

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

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

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

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

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

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

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

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

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

SOQL Operations Not Allowed with Big Objects

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



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

SEE ALSO:

[Salesforce Object Query Language \(SOQL\)](#)

[SOQL and SOSL Reference: ORDER BY](#)

CHAPTER 2 API End-of-Life Policy

See which REST API versions are supported, unsupported, or unavailable.

Salesforce is committed to supporting each API version for a minimum of 3 years from the date of first release. To improve the quality and performance of the API, versions that are over 3 years old sometimes are no longer supported.

Salesforce notifies customers who use an API version scheduled for deprecation at least 1 year before support for the version ends.

Salesforce API Versions	Version Support Status	Version Retirement Info
Versions 31.0 through 64.0	Supported.	
Versions 21.0 through 30.0	As of Summer '25, these versions are retired and unavailable.	Salesforce Platform API Versions 21.0 through 30.0 Retirement
Versions 7.0 through 20.0	As of Summer '22, these versions are retired and unavailable.	Salesforce Platform API Versions 7.0 through 20.0 Retirement

If you request any resource or use an operation from a retired API version, REST API returns the `410 : GONE` error code.

To identify requests made from old or unsupported API versions, use the [API Total Usage](#) event type.

INDEX

B

Big Object

Use cases [17](#)

Big Objects

@future [15](#)

Apex [12](#)

Composite primary key [4](#)

Custom Big Object [4](#)

Big Objects (*continued*)

Defining [4](#)

Deleting [13](#)

Deploying [4](#)

Example [15](#)

Index [4](#)

Overview [1](#), [20](#)

Populating [12](#)

Querying [21](#)