



Best Practices for Deployments with Large Data Volumes

Salesforce, Summer '25

Summer '25



CONTENTS

BEST PRACTICES FOR DEPLOYMENTS WITH LARGE DATA VOLUMES	1
Introduction	1
Underlying Concepts	2
Multitenancy and Metadata Overview	2
Search Architecture	3
Infrastructure for Systems with Large Data Volumes	4
Lightning Platform Query Optimizer	4
Database Statistics	6
Skinny Tables	7
Indexes	8
Divisions	11
Techniques for Optimizing Performance	11
Using Mashups	12
Defer Sharing Calculation	12
Using SOQL and SOSL	13
Deleting Data	15
Search	15
Best Practices	15
Reporting	16
Loading Data from the API	16
Extracting Data from the API	18
Searching	18
SOQL and SOSL	19
Deleting Data	20
General	20
Large Data Volumes Case Studies	21
Data Aggregation	21
Custom Search Functionality	22
Indexing with Nulls	22
Rendering Related Lists with Large Data Volumes	23
API Performance	23
Sort Optimization on a Query	24
Multi-Join Report Performance	24
Summary	25

BEST PRACTICES FOR DEPLOYMENTS WITH LARGE DATA VOLUMES

Introduction

Who Should Read This

This paper is for experienced application architects who work with Salesforce deployments that contain large data volumes.

A “large data volume” is an imprecise, elastic term. If your deployment has tens of thousands of users, tens of millions of records, or hundreds of gigabytes of total record storage, you have a large data volume. Even if you work with smaller deployments, you can still learn something from these best practices.

To understand the parts of this paper that deal with details of Salesforce implementation, read https://developer.salesforce.com/page/Multi_Tenant_Architecture.

Overview

Salesforce enables customers to easily scale up their applications from small to large amounts of data. This scaling usually happens automatically, but as data sets get larger, the time required for certain operations grows. The ways in which architects design and configure data structures and operations can increase or decrease those operation times by several orders of magnitude.

The main processes affected by differing architectures and configurations are:

- Loading or updating of large numbers of records, either directly or with integrations
- Extraction of data through reports and queries, or through views

The strategies for optimizing those main processes are:

- Following industry-standard practices for accommodating schema changes and operations in database-enabled applications
- Deferring or bypassing business rule and sharing processing
- Choosing the most efficient operation for accomplishing a task

What’s in This Paper

- Techniques for improving the performance of applications with large data volumes
- Salesforce mechanisms and implementations that affect performance in less-than-obvious ways
- Salesforce mechanisms designed to support the performance of systems with large data volumes

Salesforce Big Objects

Salesforce provides big data technology called 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. 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.

This paper focuses on optimizing large data volumes stored in standard and custom objects, not big objects. For optimal performance and a sustainable long-term storage solution for even larger data sets, use Bulk API or Batch Apex to move your data into big objects.

SEE ALSO:

[Salesforce Developers: Big Objects Implementation Guide](#)

Underlying Concepts

This section outlines two key concepts, multitenancy and search architecture, to explain how Salesforce:

- Provides its application to customers' instances and organizations
- Keeps supported customizations secure, self contained, and high performing
- Tracks and stores application data
- Indexes that data to optimize searching

IN THIS SECTION:

[Multitenancy and Metadata Overview](#)

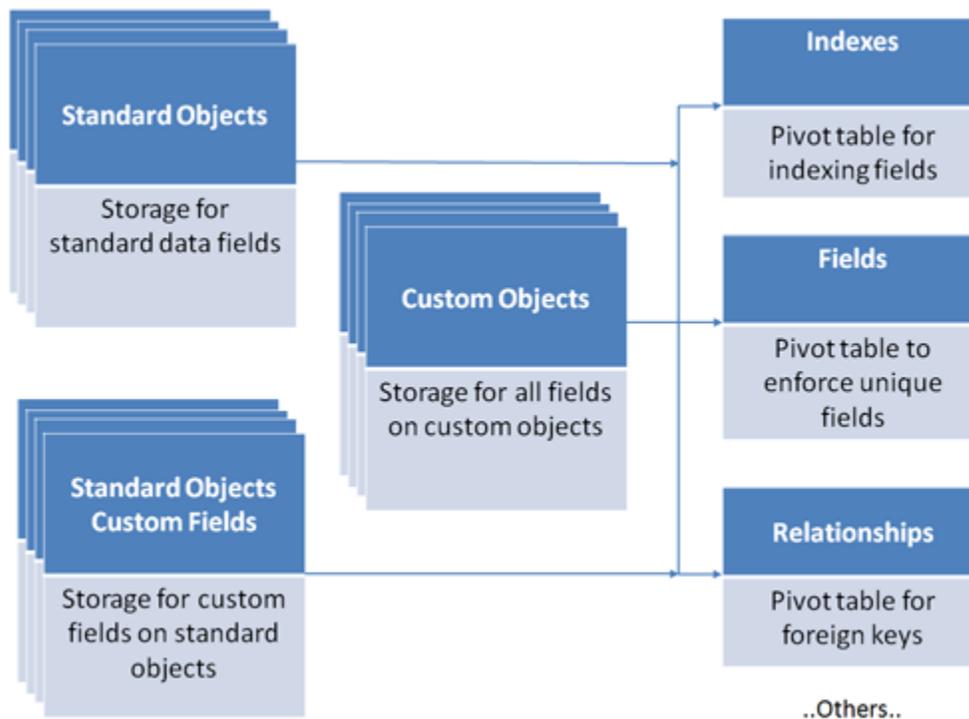
[Search Architecture](#)

Multitenancy and Metadata Overview

Multitenancy is a means of providing a single application to multiple organizations, such as different companies or departments within a company, from a single hardware-software stack. Instead of providing a complete set of hardware and software resources to each organization, Salesforce inserts a layer of software between the single instance and each organization's deployment. This layer is invisible to the organizations, which see only their own data and schemas while Salesforce reorganizes the data behind the scenes to perform efficient operations.

Multitenancy requires that applications behave reliably, even when architects are making Salesforce-supported customizations, which include creating custom data objects, changing the interface, and defining business rules. To ensure that tenant-specific customizations do not breach the security of other tenants or affect their performance, Salesforce uses a runtime engine that generates application components from those customizations. By maintaining boundaries between the architecture of the underlying application and that of each tenant, Salesforce protects the integrity of each tenant's data and operations.

When organizations create custom objects, the platform tracks metadata about the objects and their fields, relationships, and other object definition characteristics. Salesforce stores the application data for all virtual tables in a few large database tables, which are partitioned by tenant and serve as heap storage. The platform's engine then materializes virtual table data at runtime by considering the corresponding metadata.



Instead of attempting to manage a vast, ever-changing set of actual database structures for each application and tenant, the platform storage model manages virtual database structures using a set of metadata, data, and pivot tables. Thus, if you apply traditional performance-tuning techniques based on the data and schema of your organization, you might not see the effect you expect on the actual, underlying data structures.

Note: As a customer, you also cannot optimize the SQL underlying many application operations because it is generated by the system, not written by each tenant.

Search Architecture

Search is the capability to query records based on free-form text. The Salesforce search architecture is based on its own data store, which is optimized for searching for that text.

Salesforce provides search capabilities in many areas of the application, including:

- The sidebar
- Advanced and global searches
- Find boxes and lookup fields
- Suggested Solutions and Knowledge Base
- Web-to-Lead and Web-to-Case
- Duplicate lead processing
- Salesforce Object Search Language (SOSL) for Apex and the API

For data to be searched, it must first be indexed. The indexes are created using the search indexing servers, which also generate and asynchronously process queue entries of newly created or modified data. After a searchable object's record is created or updated, it could take about 15 minutes or more for the updated text to become searchable.

Salesforce performs indexed searches by first searching the indexes for appropriate records, then narrowing down the results based on access permissions, search limits, and other filters. This process creates a *result set*, which typically contains the most relevant results. After the result set reaches a predetermined size, the remaining records are discarded. The result set is then used to query the records from the database to retrieve the fields that a user sees.

 **Tip:** Search can also be accessed with SOSL, which in turn can be invoked using the API or Apex.

Infrastructure for Systems with Large Data Volumes

This section outlines:

- Salesforce components and capabilities that directly support the performance of systems with large data volumes
- Situations in which Salesforce uses those components and capabilities
- Methods of maximizing the benefits you get from the Salesforce infrastructure

IN THIS SECTION:

[Lightning Platform Query Optimizer](#)

The Salesforce multitenant architecture uses the underlying database in such a way that the database system's optimizer can't effectively optimize search queries. The Lightning Platform query optimizer helps the database's optimizer produce effective queries by providing efficient data access in Salesforce.

[Database Statistics](#)

[Skinny Tables](#)

[Indexes](#)

[Divisions](#)

Lightning Platform Query Optimizer

The Salesforce multitenant architecture uses the underlying database in such a way that the database system's optimizer can't effectively optimize search queries. The Lightning Platform query optimizer helps the database's optimizer produce effective queries by providing efficient data access in Salesforce.

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

The Lightning Platform query optimizer works on automatically generated queries that handle reports, list views, and SOQL queries. The optimizer also handles other queries that rely on these generated queries.

Specifically, the optimizer:

- Determines the best index from which to drive the query, if possible, based on filters in the query
- Determines the best table from which to drive the query, if no good index is available
- Determines how to order the remaining tables to minimize cost
- Injects custom foreign key value tables that are required to create efficient join paths
- Influences the execution plan for the remaining joins, including sharing joins, to minimize database input and output (I/O)
- Updates statistics

Creating Efficient Queries

When you're working with large volumes of data, it's important to build efficient SOQL queries, reports, and list views, which all depend on selectivity and indexes. The Lightning Platform query optimizer determines filter condition selectivity for a SOQL query, report, or list view. With some simple SOQL queries, you can easily get the necessary statistics to determine whether a specific filter condition is selective. After you index a selective field, your queries with related filter conditions can execute more efficiently and your users can be more productive.

Review the following practical considerations for gauging the selectivity of your filter conditions.

Determine the Selectivity of Your Filter Condition

To better understand selectivity, let's use an example. You build a SOQL query, report, or list view for the Opportunity object, one of the largest objects in your org. You have a filter condition, for example, a `WHERE` clause that fetches only the rows you want from the object. Is your filter condition selective enough for the optimizer to use an available index?

With a simple SOQL query, you can quickly get the statistics that help you figure out whether a value in a given field is selective.

Use SOQL to Determine the Selectivity of a Filter Condition

Consider a query that uses a basic, unary `WHERE` clause condition.

```
SELECT Id, Name FROM Opportunity
WHERE Stagename = 'Closed Won'
```

Using the query tool of your choice, for example the Developer Console Query Editor, execute the following query to get some statistics related to your filter condition's selectivity. The following is a sample query for the `Stagename` field:

```
SELECT Stagename, COUNT(id) FROM Opportunity
GROUP BY ROLLUP (Stagename)
```

The result set shows you the distribution of records for each value of the `Stagename` picklist field, including the total number of records in the object. You now have the statistics necessary to determine the selectivity for a filter condition involving the `Stagename` field.

Determine the Selectivity of More Complex Filter Conditions

Using `GROUP BY ROLLUP` in a query similar to the previous example makes it easy to get the necessary statistics to evaluate the selectivity of various conditions.

Here's an example of a query with a more complex filter condition. This query uses a date field (`CloseDate`) along with the `AND` operator:

```
SELECT Id, Name FROM Opportunity
WHERE Stagename = 'Closed Won'
AND CloseDate = THIS_WEEK
```

You already know the statistics for the `Stagename` field from the previous query. To get the same statistics for the `CloseDate` field, grouped by week for each year, use the following query.

```
SELECT WEEK_IN_YEAR(CloseDate), CALENDAR_YEAR(CloseDate), COUNT(id)
FROM Opportunity
GROUP BY ROLLUP(WEEK_IN_YEAR(CloseDate), CALENDAR_YEAR(CloseDate))
ORDER BY CALENDAR_YEAR(CloseDate), WEEK_IN_YEAR(CloseDate)
```

The query returns statistics about the distribution of Opportunity records across every week of every year by `CloseDate`.

For filter conditions that combine two or more conditions (using `AND`), the query optimizer considers the overall filter condition selective when the filter targets less than:

- Twice the selectivity thresholds for each filter
- The selectivity thresholds for the intersection of those fields

For the third example in this topic, this means:

- `Status = 'Closed Won'` is selective ($49,899 < 150,000$)
- `CloseDate = THIS_WEEK` is selective ($\sim 3000 < 150,000$)

Overall, the filter condition is selective, for both of these reasons.

If one of the filter conditions is nonselective, for example, `Status='Closed Won'` corresponds to 250,000 records, two possibilities can make the overall filter condition selective.

- Each filter condition corresponds to less than 300,000 records (twice the selectivity thresholds for each filter)
- The intersection of `Status='Closed Won' AND CloseDate = THIS_WEEK` is less than 150,000 records.

The filter condition in the example is less than 300,000 records, so the overall condition is selective.

 **Tip:** With the `OR` operator, each filter must meet the threshold individually.

Understand the Impact of Deleted Records on Selectivity

When gathering selectivity statistics, you can incorporate or exclude deleted records using the Boolean field, `IsDeleted`. This field is available in every standard and custom object.

The previous query that uses the `ROLLUP` function collects data for all Opportunity records. It doesn't matter whether `IsDeleted` is `true` or `false`. To collect the selectivity statistics for `Opportunity.StageName` and explicitly exclude deleted records, try the following query.

```
SELECT Stagename, COUNT(id) FROM Opportunity WHERE IsDeleted=false GROUP BY Stagename
```

Make Sure Fields Have Indexes

Before putting a query, report, or list view into production, confirm that a given filter condition is selective. However, the fields that make the condition selective must have indexes to be effective. Without the necessary indexes, the query optimizer must perform a full scan to fetch the target rows. The index can help make your query execute faster and improve the productivity of your org's users.

The [Database Query & Search Optimization Cheat Sheet](#) lists the standard fields that have an index by default, including `Id`, `Name`, `OwnerId`, `CreatedDate`, `SystemModstamp`, and `RecordType`, as well as all master-detail and lookup fields.

If your filter condition involves a custom field, work with Customer Support to create a custom index on the field your filter uses. Not all fields can have an index, such as non-deterministic formula fields.

Database Statistics

Modern databases gather statistics on the amount and types of data stored inside of them, and they use this information to execute queries efficiently. Because of Salesforce's multitenant approach to software architecture, the platform must keep its own set of statistical information to help the database understand the best way to access the data. As a result, when large amounts of data are created, updated, or deleted using the API, the database must gather statistics before the application can efficiently access data. Currently, this statistics-gathering process runs on a nightly basis.

Skinny Tables

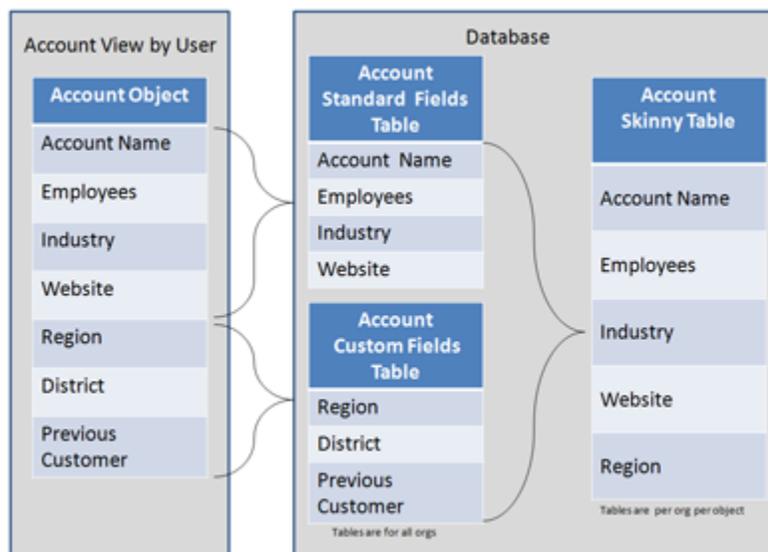
Salesforce can create *skinny tables* to contain frequently used fields and to avoid joins. This can improve the performance of certain read-only operations. Skinny tables are kept in sync with their source tables when the source tables are modified.

If you want to use skinny tables, contact Salesforce Customer Support. When enabled, skinny tables are created and used automatically where appropriate. You can't create, access, or modify skinny tables yourself. If the report, list view, or query you're optimizing changes—for example, to add new fields—you'll need to contact Salesforce to update your skinny table definition.

How Skinny Tables Can Improve Performance

For each object table that's visible to you, Salesforce maintains other, separate tables at the database level for standard and custom fields. This separation, which is invisible to customers, ordinarily requires a join when a query contains both kinds of fields. A skinny table contains both kinds of fields and also omits soft-deleted records.

This table shows an Account view, a corresponding database table, and a skinny table that can speed up Account queries.



Read-only operations that reference only fields in a skinny table don't require an extra join, and can consequently perform better. Skinny tables are most useful with tables containing millions of records to improve the performance of read-only operations, such as reports.

Important: Skinny tables aren't a magic wand to wave at performance problems. There's overhead in maintaining separate tables that hold copies of live data. Using them in an inappropriate context can lead to performance degradation instead of improvement.

Skinny tables can be created on custom objects, and on Account, Contact, Opportunity, Lead, and Case objects. They can enhance performance for reports, list views, and SOQL.

Skinny tables can contain the following types of fields.

- Checkbox
- Currency
- Date
- Date and time
- Email
- Number

- Percent
- Phone
- Picklist (multi-select)
- Text
- Text area
- Text area (long)
- URL

Skinny tables and skinny indexes can also contain encrypted data.

Here is an example of how a skinny table can speed up queries. Instead of using a date range like `01/01/11` to `12/31/11`—which entails an expensive, repeated computation to create an annual or year-to-date report—you can use a skinny table to include a `Year` field and to filter on `Year = '2011'`.

Considerations

- Skinny tables can contain a maximum of 200 columns.
- Skinny tables can't contain fields from other objects.
- For Full sandboxes: Skinny tables are copied to your Full sandbox orgs.

For other types of sandboxes: Skinny tables aren't copied to your sandbox organizations. To have production skinny tables activated for sandbox types other than Full sandboxes, contact Salesforce Customer Support.

Indexes

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

Salesforce supports custom indexes to speed up queries. You can create custom indexes by contacting Salesforce Customer Support or by deploying a custom index XML file via the [Metadata API](#).

 **Note:** The custom indexes that Salesforce Customer Support creates in your production environment are copied to all sandboxes that you create from that production environment.

The platform maintains indexes on the following fields for most objects.

- `RecordTypeId`
- `Division`
- `CreatedDate`
- `Systemmodstamp` (`LastModifiedDate`)
- `Name`
- `Email` (for contacts and leads)
- Foreign key relationships (lookups and master-detail)
- The unique Salesforce record ID, which is the primary key for each object

Salesforce also supports custom indexes on custom fields, except for multi-select picklists, text areas (long), text areas (rich), non-deterministic formula fields, and encrypted text fields.

External IDs cause an index to be created on that field. The query optimizer then considers those fields.

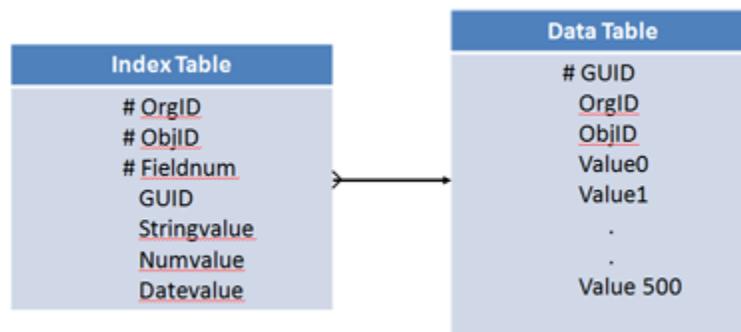
You can create External IDs only on the following fields.

- Auto Number
- Email
- Number
- Text

To create custom indexes for other field types, including standard fields, contact Salesforce Customer Support.

Index Tables

The Salesforce multitenant architecture makes the underlying data table for custom fields unsuitable for indexing. To overcome this limitation, the platform creates an index table that contains a copy of the data, along with information about the data types.



The platform builds a standard database index on this index table. The index table places upper limits on the number of records that an indexed search can effectively return.

By default, the index tables don't include records that are null (records with empty values). You can work with Salesforce Customer Support to create custom indexes that include null rows. Even if you already have custom indexes on your custom fields, you must explicitly enable and rebuild them to get the empty-value rows indexed.

Standard and Custom Indexed Fields

The query optimizer maintains a table containing statistics about the distribution of data in each index. It uses this table to perform pre-queries to determine whether using the index can speed up the query.

For example, assume that the Account object has a field called `Account_Type`—which can take the value `Large`, `Medium`, or `Small`—and that the field has a custom index.

For example, Salesforce generates a query like:

```
SELECT *
FROM Account
WHERE Account_Type__c = 'Large'
```

The query optimizer performs a pre-query to its internal statistics table to determine the number of records with `Large` in the `Account_Type` field. If this number exceeds 10% of the object's total records or 333,333 records, the query doesn't use the custom index.

The query optimizer determines what an index is used with.

Standard Indexed Fields

Used if the filter matches less than 30% of the first million records and less than 15% of additional records, up to a maximum of one million records.

For example, a standard index is used if:

- A query is executed against a table with 2 million records, and the filter matches 450,000 or fewer records.
- A query is executed against a table with 5 million records, and the filter matches 900,000 or fewer records.

Custom Indexed Fields

Used if the filter matches less than 10% of the total records, up to a maximum of 333,333 records.

For example, a custom index is used if:

- A query is executed against a table with 500,000 records, and the filter matches 50,000 or fewer records.
- A query is executed against a table with 5 million records, and the filter matches 333,333 or fewer records.

If the criteria for an indexed field aren't met, only that index is excluded from the query. If they are in the `WHERE` clause and meet the thresholds for records, other indexes are sometimes used.

The query optimizer uses similar considerations to determine whether to use indexes when the `WHERE` clause contains `AND`, `OR`, or `LIKE`.

- For `AND`, the query optimizer uses the indexes unless one of them returns more than 20% of the object's records or 666,666 total records.
- For `OR`, the query optimizer uses the indexes unless they all return more than 10% of the object's records or 333,333 total records.



Note: All fields in the `OR` clause must be indexed for any index to be used.

- For `LIKE`, the query optimizer doesn't use its internal statistics table. Instead, it samples up to 100,000 records of actual data to decide whether to use the custom index.

Custom indexes can be created on deterministic formula fields. Because some values vary over time or change when a transaction updates a related entity, the platform can't index non-deterministic formulas.

Here are examples of things that make formula fields non-deterministic.

Non-deterministic formula fields can:

- Reference other entities (like fields accessible through lookup fields)
- Include other formula fields that span over other entities
- Use dynamic date and time functions (for example, `TODAY` and `NOW`)

These formula fields are also considered non-deterministic

- Owner, autonumber, divisions, or audit fields (except for `CreatedDate` and `CreatedByID` fields)
 - References to fields that Lightning Platform can't index
 - Multi-select picklists
 - Currency fields in a multicurrency organization
 - Long text area fields
 - Binary fields (blob, file, or encrypted text)
- Standard fields with special functionalities
 - Opportunity: `Amount`, `TotalOpportunityQuantity`, `ExpectedRevenue`, `IsClosed`, `IsWon`
 - Case: `ClosedDate`, `IsClosed`
 - Product: `ProductFamily`, `IsActive`, `IsArchived`

- Solution: Status
- Lead: Status
- Activity: Subject, TaskStatus, TaskPriority

 **Note:** If the formula is modified after the index is created, the index is rebuilt.

Cross-object indexes are typically used if specified using the cross-object notation, as they are in the following example.

```
SELECT Id
FROM Score__c
WHERE CrossObject1__r.CrossObject2__r.IndexedField__c
```

You can use this approach to replace formula fields that can't be custom-indexed because they reference other objects. As long as the referenced field is indexed, the cross-object notation can have multiple levels.

Two-Column Custom Indexes

Two-column custom indexes are a specialized feature of the Salesforce platform. They're useful for list views and situations where you want to use one field to select records to display and another field to sort them. For example, an Account list view that selects by `State` and sorts by `City` can use a two-column index with `State` in the first column and `City` in the second.

When a combination of two fields is a common filter in the query string, two-column indexes typically help you sort and display records. For example, for the following SOQL, which appears in pseudo code, a two-column index on `f1__c`, `f2__c` is more efficient than single indexes on `f1__c` and `f2__c`.

```
SELECT Name
FROM Account
WHERE f1__c = 'foo'
      AND f2__c = 'bar'
```

 **Note:** Two-column indexes are subject to the same restrictions as single-column indexes, with one exception. Two-column indexes can have nulls in the second column, whereas single-column indexes can't unless Salesforce Customer Support explicitly enabled the option to include nulls.

Divisions

Divisions are a means of partitioning the data of large deployments to reduce the number of records returned by queries and reports. For example, a deployment with many customer records might create divisions called `US`, `EMEA`, and `APAC` to separate the customers into smaller groups that are likely to have few interrelationships. Divisions are beneficial for organizations with large amounts of data. To use divisions, your organization must have over one million records in a single object and more than 35 licenses.

Salesforce provides special support for partitioning data by divisions, which you can enable by contacting Salesforce Customer Support.

Techniques for Optimizing Performance

This section outlines:

- Techniques for optimizing Salesforce performance
- The arrangements, features, mechanisms, and options underpinning those techniques
- Circumstances in which you should use those techniques and tailor them to your needs

IN THIS SECTION:

- [Using Mashups](#)
- [Defer Sharing Calculation](#)
- [Using SOQL and SOSL](#)
- [Deleting Data](#)
- [Search](#)

Using Mashups

One approach to reducing the amount of data in Salesforce is to maintain large data sets in a different application, and then make that application available to Salesforce as needed. Salesforce refers to such an arrangement as a *mashup* because it provides a quick, loosely coupled integration of the two applications. Mashups use Salesforce presentation to display Salesforce-hosted data and externally hosted data.

Salesforce supports the following mashup designs.

External Website

The Salesforce UI displays an external website, and passes information and requests to it. With this design, you can make the website look like part of the Salesforce UI.

Callouts

Apex code allows Salesforce to use Web services to exchange information with external systems in real time.

Because of their real-time restrictions, mashups are limited to short interactions and small amounts of data.

See [Apex Developer Guide](#).

Advantages of Using Mashups

- Data is never stale.
- No proprietary method needs to be developed to integrate the two systems.

Disadvantages of Using Mashups

- Accessing data takes more time.
- Functionality is reduced. For example, reporting and workflow do not work on the external data.

Defer Sharing Calculation

In some circumstances, it might be appropriate to use a feature called *defer sharing calculation*, which allows users to defer the processing of sharing rules until after new users, rules, and other content have been loaded.

An organization's administrator can use a defer sharing calculation permission to suspend and resume sharing calculations, and to manage two processes: group membership calculation and sharing rule calculation. The administrator can suspend these calculations when performing a large number of configuration changes, which might lead to very long sharing rule evaluations or timeouts, and resume calculations during an organization's maintenance period. This deferral can help users process a large number of sharing-related configuration changes quickly during working hours, and then let the recalculation process run overnight between business days or over a weekend.

Using SOQL and SOSL

A SOQL query is the equivalent of a `SELECT` SQL statement, and a SOSL query is a programmatic way of performing a text-based search.

	SOQL	SOSL
Executes with	Database	Search indexes
Uses the	<code>query()</code> call	<code>search()</code> call

Use SOQL when:

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

Use SOSL when:

- You don't know in which object or field the data resides, and you want to find it in the most efficient way possible.
- You want to:
 - Retrieve multiple objects and fields efficiently, and the objects might or might not be related to one another
 - Retrieve data for a particular division in an organization using the divisions feature, and you want to find it in the most efficient way possible

Consider the following when using SOQL or SOSL.

- Both SOQL `WHERE` filters and SOSL search queries can specify text you should look for. When a given search can use either language, SOSL is generally faster than SOQL if the search expression uses a `CONTAINS` term.
- SOSL can tokenize multiple terms within a field (for example, multiple words separated by spaces) and builds a search index off this. If you're searching for a specific distinct term that you know exists within a field, you might find SOSL is faster than SOQL for these searches. For example, you might use SOSL if you were searching for "John" against fields that contained values like "Paul and John Company".
- In some cases, when multiple `WHERE` filters are being used in SOQL, indexes can't be used even though the fields in the `WHERE` clause can be indexed. In this situation, decompose the single query into multiple queries, each of which should have one `WHERE` filter, and then combine the results.
- Executing a query with a `WHERE` filter that has null values for picklists or foreign key fields doesn't use the index, and should be avoided.

For example, the following customer query performs poorly.

```
SELECT Contact__c, Max_Score__c, CategoryName__c, Category__Team_Name__c
FROM Interest__c
WHERE Contact__c != null
      AND Contact__c IN :contacts
      AND override__c != 0
      AND (
```

```

        (override__c != null AND override__c > 0)
    OR
        (score__c != null AND score__c > 0)
    )
    AND Category__c != null
    AND (
        (Category_Team_IsActive__c = true OR CategoryName__c IN :selectvalues)
        AND
        (
            Category_Team_Name__c != null
            AND
            Category_Team_Name__c IN :selectTeamValues
        )
    )
)

```

(Items preceded by colons, such as `:contacts`, are Apex variables. See [Using Apex Variables in SOQL and SOSL Queries](#) in the *Apex Developer Guide*.) Nulls in the criteria prevented the use of indexes, and some of the criteria were redundant causing the execution time to increase. Design the data model so that it doesn't rely on nulls as valid field values.

The query can be rewritten as:

```

SELECT Contact__c, Max_Score__c, CategoryName__c, Category_Team_Name__c
FROM Interest__c
WHERE Contact__c IN :contacts
    AND (override__c > 0 OR score__c > 0)
    AND Category__c != 'Default'
    AND (
        (Category_Team_IsActive__c = true OR CategoryName__c IN :selectvalues)
        AND
        Category_Team_Name__c IN :selectTeamValues
    )

```

For the field `Category__c`, the value `Default` is substituted for `NULL`, allowing an index to be used for that field.

As another example, if dynamic values are being used for the `WHERE` field, and null values can be passed in, don't let the query run to determine there are no records; instead, check for nulls and avoid the query, if possible.

A query to retrieve an account by its foreign key account number can look like this (in pseudo code).

```

SELECT Name
FROM Account
WHERE Account_ID__c = :acctid;

if (rows found == 0) return "Not Found"

```

If `acctid` is null, the entire `Account` table is scanned row by row until all data is examined.

It's better to rewrite the code as:

```

if (acctid != null) {
    SELECT Name
    FROM Account
    WHERE Account_Id__c = :acctid
}
else {
    return "Not Found"
}

```

- When designing custom query-search user interfaces, it's important to:
 - Keep the number of fields to be searched or queried to a minimum. Using many fields leads to many permutations, which can be difficult to tune.
 - Determine whether SOQL, SOSL, or a combination of the two is appropriate for the search.

Deleting Data

The Salesforce data deletion mechanism can have a profound effect on the performance of large data volumes. Salesforce uses a *Recycle Bin* metaphor for data that users delete. Instead of removing the data, Salesforce flags the data as deleted and makes it visible through the Recycle Bin. This process is called *soft deletion*. While the data is soft deleted, it still affects database performance because the data is still resident, and deleted records have to be excluded from any queries.

The data stays in the Recycle Bin for 15 days, or until the Recycle Bin grows to a specific size. The data is then *hard deleted* from the database after 15 days; when the size limit is reached; or when the Recycle Bin is emptied using the UI, the API, or Apex.

In addition, Bulk API and Bulk API 2.0 support a *hard delete* option, which allows records to bypass the Recycle Bin and become immediately available for deletion. We recommend that you use the Bulk API 2.0's hard delete function to delete large data volumes.

If you want to delete records in a sandbox organization's custom objects immediately, you can try to *truncate* those custom objects. You can contact Salesforce Customer Support for assistance with this task.

Search

When large volumes of data are added or changed, the search system must index that information before it becomes available for search by all users, and this process might take a long time.

See [Search Architecture](#) on page 3.

Best Practices

This section lists best practices for achieving good performance in deployments with large data volumes.

The main approaches to performance tuning in large Salesforce deployments rely on reducing the number of records that the system must process. If the number of retrieved records is sufficiently small, the platform might use standard database constructs like indexes or de-normalization to speed up the retrieval of data.

Approaches for reducing the number of records include:

- Reducing scope by writing queries that are narrow or selective
For example, if the Account object contains accounts distributed evenly across all states, then a report that summarizes accounts by cities in a single state is much broader—and takes longer to execute—than a report that summarizes accounts by a single city in a single state.
- Reducing the amount of data kept active
For example, if your volume of data is increasing, performance can degrade as time goes by. A policy of archiving or discarding data at the same rate at which it comes into the system can prevent this effect.

These tables feature major goals and the best practices to follow to achieve those goals.

IN THIS SECTION:

[Reporting](#)

[Loading Data from the API](#)[Extracting Data from the API](#)[Searching](#)[SOQL and SOSL](#)[Deleting Data](#)[General](#)

Reporting

Goal	Best Practice
Maximizing reporting performance by: <ul style="list-style-type: none"> • Partitioning data to match its likely use • Minimizing the number of records per object 	Reduce the number of records to query—use a value in the data to segment the query. For example, query for only a single state instead of for all states. (See Divisions on page 11.)
Reducing the number of joins	<ul style="list-style-type: none"> • Minimize the number of: <ul style="list-style-type: none"> – Objects queried in the report – Relationships used to generate the report • De-normalize data when practical—“over de-normalizing” the data results in more overhead. Use summarized data stored on the parent record for the report. This practice is more efficient than having the report summarize the child records.
Reducing the amount of data returned	Reduce the number of fields queried—only add fields to a report, list view, or SOQL query that is required.
Reducing the number of records to query	<ul style="list-style-type: none"> • Reduce the amount of data by archiving unused records—move unused records to a custom object table to reduce the size of the report object. • Use report filters that emphasize the use of standard or custom indexed fields. Use index fields in report filters, whenever possible.

Loading Data from the API

Goal	Best Practice
Improving performance	Any data operation that includes more than 2,000 records is a good candidate for Bulk API 2.0 to successfully prepare, execute, and manage an <i>asynchronous</i> workflow that makes use of the Bulk framework. Jobs with fewer than 2,000 records should involve “bulkified” <i>synchronous</i> calls in REST (for example, Composite) or SOAP.

Goal	Best Practice
Using the most efficient operations	<ul style="list-style-type: none"> Use the fastest operation possible—<code>insert()</code> is fastest, <code>update()</code> is next, and <code>upsert()</code> is next after that. If possible, also break <code>upsert()</code> into two operations: <code>create()</code> and <code>update()</code>. Ensure that data is clean before loading when using the Bulk API 2.0. Errors in batches trigger single-row processing for that batch, and that processing heavily impacts performance.
Reducing data to transfer and process	When updating, send only fields that have changed (delta-only loads).
Reducing transmission time and interruptions	For custom integrations: <ul style="list-style-type: none"> Authenticate once per load, not on each record. Use GZIP compression and HTTP keep-alive to avoid drops during lengthy save operations.
Avoiding unnecessary overhead	For custom integrations, authenticate once per load, not on each record.
Avoiding computations	Use Public Read/Write security during initial load to avoid sharing calculation overhead
Reducing computations	<ul style="list-style-type: none"> If possible for initial loads, populate roles before populating sharing rules. <ol style="list-style-type: none"> Load users into roles. Load record data with owners, triggering calculations in the role hierarchy. Configure public groups and queues, and let those computations propagate. Add sharing rules one at a time, letting computations for each rule finish before adding the next one. If possible, add people and data before creating and assigning groups and queues. <ol style="list-style-type: none"> Load the new users and new record data. Optionally, load new public groups and queues. Add sharing rules one at a time, letting computations for each rule finish before adding the next one.
Deferring computations and speeding up load throughput	Disable Apex triggers, workflow rules, and validations during loads; investigate the use of batch Apex to process records after the load is complete.

Goal	Best Practice
Balancing efficient batch sizes against potential timeouts	When using the SOAP API, use as many batches as possible—up to 200—that still avoid network timeouts if: <ul style="list-style-type: none"> Records are large. Save operations entail a lot of processing that cannot be deferred.
Optimizing the Lightning Platform Web Service Connector (WSC) to work with Salesforce	Use WSC instead of other Java API clients, like Axis.
Minimizing parent record-locking conflicts	When changing child records, group them by parent—group records by the field <code>ParentId</code> in the same batch to minimize locking conflicts.
Deferring sharing calculations	Use the defer sharing calculation permission to defer sharing calculations until after all data has been loaded. (See Defer Sharing Calculation on page 12.)
Avoiding loading data into Salesforce	Use mashups to create coupled integrations of applications. (See Using Mashups on page 12.)

Extracting Data from the API

Goal	Best Practice
Using the most efficient operations	<ul style="list-style-type: none"> Use the <code>getUpdated()</code> and <code>getDeleted()</code> SOAP API to sync an external system with Salesforce at intervals greater than 5 minutes. Use the outbound messaging feature for more frequent syncing. When using a query that can return more than one million results, consider using the query capability of the Bulk API 2.0, which might be more suitable.

Searching

Goal	Best Practice
Reducing the number of records to return	Keep searches specific and avoid using wildcards, if possible. For example, search with <code>Michael</code> instead of <code>Mi*</code> .
Reducing the number of joins	Use single-object searches for greater speed and accuracy.
Improving efficiency	Use the setup area for searching to enable language optimizations, and turn on enhanced lookups and auto-complete for better performance on lookup fields.

Goal	Best Practice
Improving search performance	In some cases, partition data with divisions. (See Divisions on page 11.)
Reducing the time it takes to index inserts and updates for large data volumes	See Search Architecture on page 3.

SOQL and SOSL

GOAL	Best Practice
Allow indexed searches when SOQL queries with multiple <code>WHERE</code> filters can't use indexes	Decompose the query. If you use two indexed fields joined by an <code>OR</code> in the <code>WHERE</code> clause, your search results can exceed the index threshold. Break the query into two queries and join the results.
Avoid querying on formula fields, which are computed in real time	If you have to query on formula fields, use formulas. Avoid filtering with formula fields that contain dynamic, non-deterministic references. See Standard and Custom Indexed Fields on page 9.
Use the most appropriate language, SOQL or SOSL, for a given search	See Using SOQL and SOSL on page 13.
Execute queries with null values in a <code>WHERE</code> filter for picklists or foreign key fields	Use values such as <code>NA</code> to replace <code>NULLS</code> options. (See Using SOQL and SOSL on page 13.)
Design efficient custom query and search user interfaces	Use SOQL and SOSL where appropriate, keep queries focused, and minimize the amount of data being queried or searched. (See Using SOQL and SOSL on page 13.)
Build efficient SOQL and SOSL queries	<p>Use filters and specific terms in your queries.</p> <p>For SOQL:</p> <ul style="list-style-type: none"> • Use selective filters, which reduce the number of rows the Query Optimizer has to scan. For example, use filters that reference indexed fields and fields that have a wider range of possible values. If filters aren't selective, the optimizer doesn't use indexed columns. • When filtering on <code>FirstName</code> and <code>LastName</code>, use the <code>Name</code> field instead. For example, <code>Select id, Email from Lead where Name='Sam Kennedy'</code> • Avoid negative filters. For example, <code>status != 'failed'</code> or <code>status != NULL</code> • Use <code>IN</code> instead of a large list of <code>OR</code> statements. For example, <code>id in ('001xxx', '001xy', '001xxz')</code>. • Avoid cross-object reference formula fields. Don't filter on these objects. They aren't indexable. <p>For SOSL:</p>

GOAL**Best Practice**

GOAL	Best Practice
Avoid timeouts on large SOQL queries	<ul style="list-style-type: none"> • For SOSL, selective filters reduce the number of irrelevant results. If the filters aren't selective and there are search terms matches for more than 2,000 records, results can be impacted by search crowding. • Don't index custom objects that you don't want searched. It adds to the number of records available to search, which can lead to search crowding. • Filter out objects you don't want searched. • Use specific search terms. • Use targeted search groups. Search groups include NAME, EMAIL, and PHONE fields. For example, <code>FIND 'Avery Smith' IN NAME FIELDS RETURNING Account (Id, Name) , Lead (Id, Name)</code>
	<p>Tune the SOQL query, reducing query scope, and using selective filters. Consider using Bulk API 2.0 with Bulk API 2.0 Query. If you've used the previous suggestions and still get timeouts, consider adding a LIMIT clause (starting with 100,000 records) to your queries. If using batch Apex for your queries, use chaining to get sets of records (using LIMIT) or consider moving filter logic to the execute method.</p>

Deleting Data

Goal**Best Practice**

Deleting large volumes of data	When deleting large volumes of data, a process that involves deleting one million or more records, use the hard delete option of the Bulk API or Bulk API 2.0 . Deleting large volumes of data might take significant time due to the complexity of the deletion process. (See Deleting Data on page 20.)
Making the data deletion process more efficient	When deleting records that have many children, delete the children first.

General

Goal**Best Practice**

Avoiding sharing computations	Avoid having any user own more than 10,000 records.
Improving performance	Use a data-tiering strategy that spreads data across multiple objects, and brings in data on demand from another object or external store.

Goal	Best Practice
Reducing the time it takes to create full copies of production sandboxes with large data volumes	When creating copies of production sandboxes, exclude field history if it isn't required, and don't change a lot of data until the sandbox copy is created.
Making deployments more efficient	Distribute child records so that no parent has more than 10,000 child records. For example, in a deployment that has many contacts but does not use accounts, set up several dummy accounts and distribute the contacts among them.

Large Data Volumes Case Studies

This section contains:

- Large data volume-related problems that customers have had
- Solutions that customers used—or could have used—to fix those problems

To recognize and solve similar issues, read the following case studies:

IN THIS SECTION:

- [Data Aggregation](#)
- [Custom Search Functionality](#)
- [Indexing with Nulls](#)
- [Rendering Related Lists with Large Data Volumes](#)
- [API Performance](#)
- [Sort Optimization on a Query](#)
- [Multi-Join Report Performance](#)

Data Aggregation

Situation

The customer needed to aggregate monthly and yearly metrics using standard reports. The customer's monthly and yearly details were stored in custom objects with four million and nine million records, respectively. The reports were aggregating across millions of records across the two objects, and performance was less than optimal.

Solution

The solution was to create an aggregation custom object that summarized the monthly and yearly values into the required format for the required reports. The reports were then executed from the aggregated custom object. The summarization object was populated using batch Apex.

Custom Search Functionality

Situation

The customer needed to search in large data volumes across multiple objects using specific values and wildcards. The customer created a custom Visualforce page that would allow the user to enter 1–20 different fields, and then search using SOQL on those combinations of fields.

Search optimization became difficult because:

- When many values were entered, the `WHERE` clause was large and difficult to tune. When wildcards were introduced, the queries took longer.
- Querying across multiple objects was sometimes required to satisfy the overall search query. This practice resulted in multiple queries occurring, which extended the search.
- SOQL is not always appropriate for all query types.

Solutions

The solutions were to:

- Use only essential search fields to reduce the number of fields that could be searched. Restricting the number of simultaneous fields that could be used during a single search to the common use cases allowed Salesforce to tune with indexes.
- De-normalize the data from the multiple objects into a single custom object to avoid having to make multiple querying calls.
- Dynamically determine the use of SOQL or SOSL to perform the search based on both the number of fields searched and the types of values entered. For example, very specific values (i.e., no wild cards) used SOQL to query, which allowed indexes to enhance performance.

Indexing with Nulls

Situation

The customer needed to allow nulls in a field and be able to query against them. Because single-column indexes for picklists and foreign key fields exclude rows in which the index column is equal to null, an index could not have been used for the null queries.

Solution

The best practice would have been to not use null values initially. If you find yourself in a similar situation, use some other string, such as `N/A`, in place of `NULL`. If you cannot do that, possibly because records already exist in the object with null values, create a formula field that displays text for nulls, and then index that formula field.

For example, assume the `Status` field is indexed and contains nulls.

Issuing a SOQL query similar to the following prevents the index from being used.

```
SELECT Name
FROM Object
WHERE Status__c = ''
```

Instead, you can create a formula called `Status_Value`.

```
Status_Value__c = IF(ISBLANK(Status__c), "blank", Status__c)
```

This formula field can be indexed and used when you query for a null value.

```
SELECT Name
FROM Object
WHERE Status_Value__c = 'blank'
```

This concept can be extended to encompass multiple fields.

```
SELECT Name
FROM Object
WHERE Status_Value__c = '' OR Email = ''
```

For more information about standard and custom indexed fields, see [Indexes](#) in *Best Practices for Deployments with Large Data Volumes*.

To avoid long execution times of SOQL queries, see [Improve Performance of SOQL Queries using a Custom Index](#).

Rendering Related Lists with Large Data Volumes

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

Situation

The customer had hundreds of thousands of account records and 15 million invoices, which were within a custom object in a master-detail relationship with the account. Each account record took a long time to display because of the Invoices related list's lengthy rendering time.

Solution

The delay in displaying the Invoices related list was related to data skew. While most account records had few invoice records, there were some records that had thousands of them.

To reduce the delay, the customer tried to reduce the number of invoice records for those parents and keep data skew to a minimum in child objects. Using the `Enable Separate Loading of Related Lists` setting allowed the account detail to render while the customer was waiting for the related list query to complete.

API Performance

Situation

The customer designed a custom integration to synchronize Salesforce data with external customer applications.

The integration process involved:

- Querying Salesforce for all data in a given object
- Loading this data into the external systems
- Querying Salesforce again to get IDs of all the data, so the integration process could determine what data had been deleted from Salesforce

The objects contained several million records. The integration also used a specific API user that was part of the sharing hierarchy to limit the records retrieved. The queries were taking minutes to complete.

In Salesforce, sharing is a very powerful mechanism for making certain records visible to a certain user, and it works very well for UI interactions. However, when used as a high-volume data filter in a SOQL query, performance can suffer because data access is more complex and difficult to process when you use sharing as a filter, especially if you are trying to filter out records in a large data volume situation.

Solution

The solution was to give the query access to all the data, and then to use selective filters to get the appropriate records. For example, using an administrator as the API user would have provided access to all of the data and prevented sharing from being considered in the query.

An additional solution would have been to create a delta extraction, lowering the volume of data that needed to be processed.

You can find more information about how sharing can affect performance in [Platform Sharing Architecture](#).

Sort Optimization on a Query

Situation

The customer had the following query.

```
SELECT Id,Product_Code__c
FROM Customer_Product__c
WHERE CreatedDate = Last_N_Days:3
```

The query was looking for all the records created in the last three days, but the amount of data in the object exceeded the threshold for standard indexes: 30% of the total records up to one million records. The query performed poorly.

Solution

The query was rewritten as:

```
SELECT Id,Product_Code__c
FROM Customer_Product__c
WHERE CreatedDate = Last_N_Days:3
ORDER BY CreatedDate LIMIT 99999
```

In this query, the threshold checks were not done, and the `CreatedDate` index was used to find the records. This kind of query returns a maximum of 99,999 records in the order that they were created within the last three days, assuming that 99,999 or fewer records were created during the last three days.

 **Note:** In general, when querying for data that has been added over the `Last_N_Days`, if you specify an `ORDER BY` query on an indexed field with a limit of fewer than 100,000 records, the `ORDER BY` index is used to do the query.

Multi-Join Report Performance

Situation

The customer created a report that used four related objects: Accounts (314,000), Sales Orders (769,000), Sales Details (2.3 million), and Account Ownership (1.2 million). The report had very little filtering and needed to be optimized.

Solution

To optimize the report, the customer:

- Added additional filters to make the query more selective and ensured that as many filters as possible were indexable
- Reduced the amount of data in each object, whenever possible
- Kept the Recycle Bin empty. Data in the Recycle Bin affects query performance.
- Ensured that no complex sharing rules existed for the four related objects. Complex sharing rules can have a noticeable impact on performance.

Summary

The Salesforce platform is a robust environment in which native and custom applications can scale to large volumes of data very quickly while continuing to perform well.

You can maximize the benefits of these capabilities by:

- Making queries selective—ensure that reports, list views, and SOQL are using appropriate filters.
- Reducing the amount of active data—use archiving, mashups, and other techniques to reduce the amount of data stored in Salesforce.

Following these two broad principles and the best practices that support them can reduce the impact of large data volumes on Salesforce application performance.