Netezza to BigQuery migration guide
# About this document

Objectives of this guide .......................... 2
Nonobjectives of this guide ................. 2

## Introduction

Premigration .................................................. 3
- BigQuery capacity planning ............... 4
- Security in Google Cloud .................. 4
  - Identity and access management ....... 4
  - Row-level security ................. 6
  - Data encryption ..................... 8
    - Performance benchmarking .... 8
    - Foundational project setup .... 9
    - Network connectivity ...... 9

Migration to Google Cloud: An in-depth look .......................... 9
- Architecture comparison ............... 9
  - Netezza architecture ............. 9
  - BigQuery architecture .......... 11
- Detailed migration steps ........... 13
  - Exporting data out of Netezza .... 13
  - Data staging (in Cloud Storage) .... 13
  - Loading data directly into BigQuery .... 14
- Data type mapping in BigQuery ...... 15
- SQL comparison .................... 16
- Function comparison .......... 16

Postmigration ............................................. 16
- Connecting to BigQuery APIs over the internet .... 17
- Optimizing for BigQuery .............. 17
  - Replacing the Months_Between function in BigQuery with UDFs .... 18
  - Netezza stored procedure migration .... 18
  - Custom UDF to emulate Netezza ASCII .... 18
About this document

The information and recommendations in this document were gathered through work with a variety of clients and environments in the field.

<table>
<thead>
<tr>
<th>What’s covered</th>
<th>Migration from Netezza to Google Cloud</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary audience</td>
<td>Enterprise architects, DBAs, application developers, IT security</td>
</tr>
<tr>
<td>IT environment</td>
<td>Netezza, Google Cloud</td>
</tr>
<tr>
<td>Deployment phases</td>
<td>● Exporting Netezza to Cloud Storage</td>
</tr>
<tr>
<td></td>
<td>● Ingesting from Cloud Storage into BigQuery</td>
</tr>
<tr>
<td></td>
<td>● Data modeling</td>
</tr>
<tr>
<td></td>
<td>● Third-party tools</td>
</tr>
<tr>
<td>Takeaways</td>
<td>● Strategies for migration</td>
</tr>
<tr>
<td></td>
<td>● Query optimization and data modeling best practices</td>
</tr>
<tr>
<td></td>
<td>● Troubleshooting tips</td>
</tr>
<tr>
<td></td>
<td>● User adoption guidance</td>
</tr>
</tbody>
</table>

Objectives of this guide

● Provide high-level guidance to organizations migrating from Netezza to BigQuery.
● Show ways organizations can rethink their existing data model and extract, transform, and load (ETL) processes to get the most out of BigQuery.
● Demonstrate new capabilities available to your organization through BigQuery, not map features one-to-one with Netezza.

Nonobjectives of this guide

● Provide detailed instructions for all activities required to migrate from Netezza to BigQuery.
● Present solutions for every use case of the Netezza platform in BigQuery.
● Replace the BigQuery documentation or act as a substitute for training on big data. We strongly recommend that you take the Big Data and Machine Learning classes and review the extensive documentation prior to, or in conjunction with, this guide.
● Provide guidance for integrating consuming analytic applications like MicroStrategy with BigQuery. See existing documentation about the vendor portal, including How to connect to Google BigQuery.

Introduction

For decades, large organizations have relied on systems like Netezza to help store and analyze massive amounts of data. While these systems are powerful, they require huge investments in hardware, maintenance, and licensing. In addition, as the number of data sources and the volume of data increases, organizations face challenges around node management, volume of data per source, archiving costs, and overall scalability of the system.

As a result, more and more organizations are evaluating BigQuery to solve their need for a cloud-based enterprise data warehouse. BigQuery is Google’s fully managed, petabyte-scale, serverless enterprise data warehouse for analytics. With BigQuery, there is no infrastructure to manage, and you don’t need a database administrator. You can focus on analyzing data to find meaningful insights using familiar SQL.

BigQuery can scan billions of rows, without an index, in tens of seconds. BigQuery is a cloud-powered, massively parallel query service that shares Google’s infrastructure, so it can parallelize each query and run it on tens of thousands of servers simultaneously. The two core technologies that differentiate BigQuery are columnar storage and tree architecture:

● **Columnar storage:** Data is stored in columns rather than rows, which makes it possible to achieve a very high compression ratio and scan throughput.

● **Tree architecture:** Queries are dispatched and results are aggregated across thousands of machines in a few seconds.

The technical architecture of BigQuery is explained in more detail in An Inside Look at Google BigQuery.

Premigration

To ensure a successful data warehouse migration, start planning your migration strategy early in your project timeline. This lets you evaluate the Google Cloud features that suit your needs.

For more details, see Migrating Teradata to BigQuery.
BigQuery capacity planning

Analytics throughput in BigQuery is measured in slots. A BigQuery slot is Google’s proprietary unit of compute, RAM, and network throughput required to execute SQL queries. BigQuery automatically calculates how many slots are required by each query, depending on the query size and complexity.

The most common comparison customers make to a BigQuery slot is Netezza’s Snippet Processing Units (SPUs). Although a slot and an SPU do share some similarities (for example, CPU, memory, and processing of data), they do not represent the same unit of measure. Rather, to help with slot estimation, we recommend setting up BigQuery monitoring using Cloud Monitoring and analyzing your audit logs using BigQuery. Many customers use tools like Google Data Studio or Tableau to visualize BigQuery slot utilization. Regularly monitoring and analyzing your slot utilization helps you estimate how many total slots your organization needs as you grow on Google Cloud.

For example, suppose you initially reserve 2,000 BigQuery slots to run 50 medium-complexity queries simultaneously. If you notice queries consistently taking more than a few hours to run and your dashboards show high slot utilization, your queries might not be optimized or you might need additional BigQuery slots to help support your workloads. If you want to purchase slots yourself in flex, monthly, or yearly commitments, you can get started with BigQuery Reservations using the Google Cloud Console or the bq command-line tool. If you signed an offline agreement for your flat-rate purchase, your plan might deviate from the details described here. Contact your sales representative to take advantage of BigQuery Reservations.

A typical customer with around 300 TB of data in Netezza could incur a query cost of $2,000 per month or 120 slots. The query-to-storage cost ratio is roughly two to one.

Security in Google Cloud

The following sections describe the common Netezza security controls and how you can help protect your data warehouse in a Google Cloud environment.

Identity and access management

The Netezza database contains a set of fully integrated system access control capabilities permitting users access to the resources to which they are authorized.
The Netezza database supports the following access methods:

- **Database users and user groups:** In order to access the Netezza database, users must have Netezza database user accounts.

- **Security model:** Features a combination of administrator privileges that are granted to users and groups, plus object privileges that are associated with specific objects (such as a single table) and classes of objects (such as all tables).

- **Logon authentication:** Client encryption and security.

- **User and group limits:** Place limits on the resources that users and groups can use. You can limit the number of rows that a query can return (rowset limit), the amount of time a session can remain idle before it is terminated (session timeout), the amount of time a query can run before the system notifies you, the session priority, and the number of days before the account password expires.

- **Logging Netezza SQL information:** Log information about all user or application activity on the server as well as information that is generated by individual Windows clients.

Access to Netezza is controlled through the network to the Netezza appliance by managing the Linux user accounts that can log in to the operating system. Access to the Netezza database, objects, and tasks are managed using the Netezza database user accounts that can establish SQL connections to the system.

BigQuery uses Identity and Access Management (IAM) to manage access to resources. The types of resources available in BigQuery are organizations, projects, datasets, tables, and views. In the IAM policy hierarchy, datasets are child resources of projects. A table inherits permissions from the dataset that contains it.

To grant access to a resource, assign one or more roles to a user, group, or service account. Organization and project roles affect the ability to run jobs or manage the project, whereas dataset roles affect the ability to access or modify the data inside a project.

IAM provides these types of roles:

- **Predefined roles** are meant to support common use cases and access control patterns.

- **Primitive roles** include the Owner, Editor, and Viewer roles. Predefined roles provide granular access for a specific service and are managed by Google Cloud.

- **Custom roles** provide granular access according to a user-specified list of permissions.
When you assign both predefined and primitive roles to a user, the permissions granted are a union of the permissions of each individual role.

**Row-level security**

Multi-level security is an abstract security model, which Netezza uses to define rules to control user access to row-secure tables (RSTs). A row-secure table is a database table with security labels on rows to filter out users without the appropriate privileges. The results that are returned on queries differ based upon the privileges of the user who makes the query.

Although BigQuery does not have an explicit row-level security module, you can achieve the same functionality by using **authorized views**. Authorized views let you permit users to query a view without having read access to the underlying tables. To create row-level security over a table, you create an authorized view that returns only the rows appropriate for that security category.

If you want to give different users access to different rows in your table, without having to create an individual view for each, there are a few options. These options all make use of the `SESSION_USER()` function in BigQuery, which returns the email address of the currently running user. In this example, `example@google.com` represents that email address.

In practice, you'll want to specify a group. Your schema looks like `{customer:string, id:integer, allowed_group: string}`, and anyone in `allowed_group` can see your table. The `allowed_group` members are represented in another table that contains your group mappings. That table looks like this: `{group:string, user_name:string}`. The rows might look like this:

<table>
<thead>
<tr>
<th></th>
<th>Email Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>engineers</td>
<td><a href="mailto:example@google.com">example@google.com</a></td>
</tr>
<tr>
<td>engineers</td>
<td><a href="mailto:some_engineer@google.com">some_engineer@google.com</a></td>
</tr>
<tr>
<td>administrators</td>
<td><a href="mailto:some_admin@google.com">some_admin@google.com</a></td>
</tr>
<tr>
<td>sales</td>
<td><a href="mailto:some_salesperson@google.com">some_salesperson@google.com</a></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After creating the group mapping table, you can update the view definition defined previously as follows (where `private.access_control` holds the group mapping):

```sql
#standardSQL
SELECT c.customer, c.id
FROM private.customers c
INNER JOIN (SELECT DISTINCT group
              FROM private.access_control) gc
ON c.id = gc.id
WHERE gc.group IN (engineers, administrators, sales);
```
FROM private.access_control
WHERE user_name = SESSION_USER() g
ON c.allowed_group = g.group

If multiple access levels are needed depending on a role, the sample query for the authorized view might look like the following:

SELECT c.customer, c.id
FROM private.customers c
JOIN private.access_control g ON g.user_name = SESSION_USER()
WHERE (CASE WHEN g.role = 'Representative' THEN c.representative = g.user_name
    WHEN g.role = 'Unit Manager' THEN c.unit = g.unit
    WHEN g.role = 'Director' THEN c.department = g.department
    ELSE 1 = 0
END )

In this way, you can manage the access control group mapping (private.access_control) separately from the data table (private.customers). You can also add the ability for groups to contain other groups. You can get this by doing a more complex join to expand the groups in the access control table. Consider saving the results each time you query the main table.

There are other options like adding another column to your table for the user who is allowed to see the row. For example, the schema `{customer:string, id:integer}` would become `{customer:string, id:integer, allowed_viewer: string}`. Then you can define a view as follows:

```sql
#standardSQL
SELECT customer, id
FROM private_customers
WHERE allowed_viewer = SESSION_USER()
```

**Note:** Make sure that there are no duplicates in private.access_control. Otherwise, records might be repeated in the results.

In this way, you can manage the access control group mapping (private.access_control) separately from the data table (private.customers).
You can also add the ability for groups to contain other groups. You can get this by doing a more complex join to expand the groups in the access control table. Consider saving the results each time you query the main table.

**Data encryption**

Netezza appliances use **self-encrypting drives (SEDs)** for improved security and protection of the data stored on the appliance.

SEDs encrypt data as it is written to the disk. Each disk has a disk encryption key (DEK) that is set at the factory and stored on the disk. The disk uses the DEK to encrypt data as it writes and then to decrypt the data as it is read from disk. The operation of the disk, and its encryption and decryption, is transparent to the users who are reading and writing data. This default encryption and decryption mode is referred to as **secure erase mode**.

In secure erase mode, you don't need an authentication key or password to decrypt and read data. SEDs offer improved capabilities for an easy and speedy secure erase for situations when disks must be repurposed or returned for support or warranty reasons.

Netezza uses symmetric encryption, and the decrypt function might be relevant for encrypted databases.

```sql
varchar = decrypt(varchar text, varchar key [, int algorithm [, varchar IV]]);
nvarchar = decrypt(nvarchar text, nvarchar key [, int algorithm[, varchar IV]]);
```

All data stored within BigQuery is encrypted at rest. If you want to control encryption yourself, you can use customer-managed encryption keys (CMEK) for BigQuery. Instead of Google managing the key encryption keys that protect your data, you control and manage key encryption keys in **Cloud Key Management Service**.

**Performance benchmarking**

It is important to establish a baseline performance for the current-state Netezza environment by selecting a set of representational queries. These queries are captured from the consuming applications (such as Tableau or Cognos).

<table>
<thead>
<tr>
<th>Environment</th>
<th>Netezza</th>
<th>BigQuery</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data size</td>
<td>size TB</td>
<td>-</td>
</tr>
<tr>
<td>Query 1: <code>name</code> (full table scan)</td>
<td><code>mm:ss.ms</code></td>
<td>-</td>
</tr>
</tbody>
</table>
Foundational project setup

It is necessary to complete project setup prior to provisioning storage resources for migration of data. Refer to Best practices for enterprise operations for guidance on setting up projects and enabling IAM at the project level.

This section describes how to build the landing zone for the data and create a secure connection to it. It also offers IAM best practices.

Network connectivity

A reliable and secure network connection is required between the on-premises data center (where the Netezza instance is running) and the Google Cloud environment. Network bandwidth can be a limiting factor when uploading data extracts.

Migration to Google Cloud: An in-depth look

BigQuery is a powerful data warehouse suitable for large-scale analytics used by all types of organizations, from startups to Fortune 500 companies. Understanding some of the key features that BigQuery offers is critical to improving performance and reducing cost.

Architecture comparison

Netezza architecture

The Netezza Data Warehouse Appliances models include Skimmer (N100-1 series), TwinFin (N1001 series), Striper (N2001 series), and IBM PureData System for Analytics (N2002, N3001 series).

Netezza TwinFin and Striper models reached their end of support in June 2019.

Netezza is a hardware accelerated appliance that has the following features:

- Snippet Processing Units (SPUs), or S-Blades, are query processing engines. Each SPU has a CPU, FPGAs, memory, and I/O to process queries and query results. Each SPU has associated data partitions.
- SMP hosts are Linux servers set up in an active-passive configuration for high availability. The active host presents a standardized interface to external tools and
applications. It compiles SQL queries into executable code segments called snippets, creates optimized query plans, and distributes the snippets to the MPP nodes for execution.

- Disk enclosure consists of high-density, high-performance disks that are RAID protected. Each disk contains a slice of every database table’s data. A high-speed network connects disk enclosures to S-Blades, allowing all the disks in a Netezza system to simultaneously stream data to the S-Blades at the maximum rate possible. The Netezza system checks if a rebalance is required when the system is restarted, when a SPU failover happens, or when a disk regeneration setup fails. There are situations in which `nzhw rebalance -check` commands need to be run in order to determine if a rebalance is required. Excessive data deletion or data ingestion scenarios also need to be monitored closely to make sure the SPU and data slices are balanced.

The following diagram illustrates the data abstraction layers within the platform:
Disks: Physical drives within the disk enclosures store the databases and tables.

Data slices: Logical representation of the data that is saved on a disk. Data is distributed across the data slices using a distribution key. Status of the data slices can be monitored by issuing the `nzds show` command.

<table>
<thead>
<tr>
<th>Data slice status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Healthy</td>
<td>The data slice is operating normally and the data is protected in a redundant configuration; that is, the data is fully mirrored.</td>
</tr>
<tr>
<td>Repairing</td>
<td>The data slice is in the process of being regenerated to a spare disk because of a disk failure.</td>
</tr>
<tr>
<td>Degraded</td>
<td>The data slice is not protected in a redundant configuration. Another disk failure could result in loss of a data slice, and the degraded condition affects system performance.</td>
</tr>
</tbody>
</table>

Data partitions: Logical representation of a data slice that is managed by a specific SPU. Each SPU owns one or more data partitions containing the user data that the SPU is responsible for processing during queries.

Network fabric connects all system components. The Netezza appliance runs a customized IP-based protocol.

TwinFin is offered in a single-rack and multi-rack configuration. The following table highlights the processing and data storage capacity of the appliances.

<table>
<thead>
<tr>
<th>Specifications</th>
<th>TwinFin 3</th>
<th>TwinFin 6</th>
<th>TwinFin 12</th>
<th>2 racks</th>
<th>3+ racks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Racks</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3-10</td>
</tr>
<tr>
<td>SPU or S-Blades</td>
<td>3</td>
<td>6</td>
<td>12</td>
<td>24</td>
<td>racks × 12</td>
</tr>
<tr>
<td>CPU cores</td>
<td>24</td>
<td>48</td>
<td>96</td>
<td>192</td>
<td>racks × 96</td>
</tr>
<tr>
<td>Storage (TB)</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>racks × 32</td>
</tr>
</tbody>
</table>

BigQuery architecture

In BigQuery, the data is physically stored on Google’s distributed file system, called Colossus, which ensures durability by using erasure encoding to store redundant chunks of data on multiple physical disks. Moreover, the data is replicated to multiple data centers.
BigQuery is a platform as a service (PaaS). Its capacity scales up and down without any intervention from the user as Google manages the backend for the user. As a result, unlike with many RDBMS systems, you don’t need to provision resources before using BigQuery. BigQuery allocates storage and query resources dynamically based on your usage patterns. Storage resources are allocated as you consume them and deallocated as you remove data or drop tables. Query resources are allocated according to query type and complexity. Each query uses some number of slots, which are units of computation that comprise a certain amount of CPU and RAM. An eventual fairness scheduler is used, so there might be short periods where some queries get a disproportionate share of slots, but the scheduler eventually corrects this.

In an SQL spooling process of writing query results to a file, BigQuery creates a temporary table to store the results of every query that runs. These results can be saved to a permanent table or downloaded to a local file. During the query execution, BigQuery stores files in temporary Colossus file system locations. For more information on how a temporary result set is created during the query process, see Writing query results.

In traditional VM terms, BigQuery gives you the equivalent of both:

- Per-second billing
- Per-second scaling

To accomplish this task, BigQuery does the following:

- Keeps vast resources deployed under the hood to avoid having to rapidly scale.
- Uses multitenant resources to instantly allocate large chunks for seconds at a time.
- Efficiently allocates resources across users with economies of scale.
- Charges you only for the jobs you run, rather than for deployed resources, so you don’t pay for resources that you don’t use.
For more information on pricing, see Understanding BigQuery’s rapid scaling and simple pricing. For more information on storage and compute provisioning and sizing, see BigQuery for data warehouse practitioners.

Detailed migration steps

Exporting data out of Netezza

The following options are available for data export from the Netezza database tables in CSV format:

- **External table**: The recommended and faster option. Refer to the Netezza documentation for more details on using this option.

  ```sql
  CREATE EXTERNAL TABLE '/data/out/netezza-export.csv'
  USING (delimiter ',') AS
  SELECT column1, column2, ..., columnN
  FROM netezzza_table;
  ```

- **NZSQL**: Comparatively slower than external tables.

  ```bash
  nzsql -u dbusername -pw dbpassword -db database -h host -F "\," -t -A -o '/data/out/netezza-export.csv' -c "SELECT column1, column2, ..., columnN FROM netezza_table"
  ```

- **NZ_UNLOAD**

  ```bash
  nz_unload -db database -sql "select column1, column2, ..., columnN from netezza_table" -file "/data/out/netezza-export.txt"
  ```

Data staging (in Cloud Storage)

Cloud Storage is the recommended landing place (staging area) for data exported from Netezza. Cloud Storage is designed for fast, flexible ingestion of structured or unstructured data.
Loading data directly into BigQuery

Data can be pushed into BigQuery’s data store either directly using data load from Cloud Storage, using Dataflow, or using real-time streaming. Dataflow is used when data transformation is required.

Netezza does not currently support Avro exports, so you should export uncompressed flat files (CSV) for each table. Data can be exported using the `CREATE EXTERNAL TABLE` command in NzQL. An alternative approach is to read data using third-party systems like Informatica (or custom ETL) using JDBC/ODBC connectors to produce CSV files.

Limit file sizes to 4 TB (uncompressed) for faster loading into BigQuery. Using `gsutil`, you can automate and parallelize the transfer of the files to Cloud Storage. However, you have to export the schema beforehand. This is a good opportunity to optimize BigQuery using partitioning and clustering.

Cloud Storage is ideal for storing large amounts of data. Cloud Storage is a highly available and durable object store service with no limitations on the number of files, and you pay only for the storage you use. The service is optimized to work with other Cloud Platform services such as BigQuery and Dataflow, making it easier for you to analyze your data.

The following command copies all files from a source directory into a Cloud Storage bucket:

```
gsutil -m cp -r SOURCE_DIRECTORY gs://BUCKET_NAME
```

**Note:** The `-m` flag in `gsutil` performs the copy operation using a combination of multithreading and multiprocessing, using a number of threads and processors determined by the `parallel_thread_count` and `parallel_process_count` values set in the boto configuration file. This significantly improves performance if you are performing operations on a large number of files over a reasonably fast network connection. For more information on performance tuning large data transfers using `gsutil`, see Transferring your large data sets.

The right side of the figure shows data movement inside Google Cloud. Depending on your team’s preference for working in Java, Python, or SQL, several options are available:

- If you don’t intend to perform data transformation, you can move data directly from Cloud Storage to BigQuery.
- After exported data is loaded into BigQuery, transform it using one of the following:
  - The `bq` command-line tool to execute a series of queries to format the data.
  - Dataflow to execute transformations in ELT style.
Data type mapping in BigQuery

Netezza data types differ from BigQuery data types. For more detail on BigQuery data types, see the official [BigQuery documentation](https://cloud.google.com/bigquery/docs). For a detailed comparison between Netezza and BigQuery data types, see [Netezza-to-BigQuery SQL translation reference](https://cloud.google.com/bigquery/docs/translation-reference).

<table>
<thead>
<tr>
<th>Netezza</th>
<th>BigQuery</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER/INT/INT4</td>
<td>INT64</td>
<td></td>
</tr>
<tr>
<td>SMALLINT/INT2</td>
<td>INT64</td>
<td></td>
</tr>
<tr>
<td>BYTEINT/INT1</td>
<td>INT64</td>
<td></td>
</tr>
<tr>
<td>BIGINT/INT8</td>
<td>INT64</td>
<td></td>
</tr>
<tr>
<td>DECIMAL</td>
<td>NUMERIC</td>
<td>The NUMERIC type in BigQuery does not enforce custom digit or scale bounds (constraints) like Netezza does.</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>NUMERIC</td>
<td>The NUMERIC type in BigQuery does not enforce custom digit or scale bounds (constraints) like Netezza does.</td>
</tr>
<tr>
<td>NUMERIC(p,s)</td>
<td>NUMERIC</td>
<td></td>
</tr>
<tr>
<td>FLOAT(p)</td>
<td>FLOAT64</td>
<td></td>
</tr>
<tr>
<td>REAL/FLOAT(6)</td>
<td>FLOAT64</td>
<td></td>
</tr>
<tr>
<td>DOUBLE PRECISION/FLOAT(14)</td>
<td>FLOAT64</td>
<td></td>
</tr>
<tr>
<td>CHAR/CHARACTER</td>
<td>STRING</td>
<td>The STRING type in BigQuery is variable-length and does not require manually setting a maximum character length as the Netezza CHARACTER and VARCHAR types require.</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>STRING</td>
<td>The STRING type in BigQuery is variable-length and does not require manually setting a maximum character length as the Netezza CHARACTER and VARCHAR types require.</td>
</tr>
<tr>
<td>NCHAR</td>
<td>STRING</td>
<td>The STRING type in BigQuery is stored as variable-length UTF-8 encoded Unicode.</td>
</tr>
<tr>
<td>NVARCHAR</td>
<td>STRING</td>
<td>The STRING type in BigQuery is stored as variable-length UTF-8 encoded Unicode.</td>
</tr>
<tr>
<td>VARBINARY</td>
<td>BYTES</td>
<td></td>
</tr>
</tbody>
</table>
The **BOOL** type in BigQuery can only accept `TRUE`/`FALSE` unlike the **BOOL** type in Netezza, which can accept a variety of values like `0/1`, `yes/no`, `true/false`, `on/off`.

The **TIME** type in BigQuery is independent of a time zone.

The Netezza **TIMESTAMP** type has microsecond precision (including leap seconds) and is usually associated with UTC time zone, the same as BigQuery (details).

There is no **ARRAY** data type in Netezza. The **ARRAY** type is instead stored in a **varchar** field (details).

**SQL comparison**

Netezza data SQL consists of DDL, DML, and Netezza-only Data Control Language (DCL), which are different from BigQuery SQL. BigQuery **standard SQL** is compliant with the SQL 2011 standard and has extensions that support querying nested and repeated data. If you are using BigQuery legacy SQL, see Legacy SQL Functions and Operators. For a detailed comparison between Netezza and BigQuery SQLs and functions, see Netezza to BigQuery SQL translation reference.

**Function comparison**

It is important to understand how the Netezza functions map to BigQuery functions. For example, the `MONTHS_BETWEEN` function in Netezza outputs a decimal point, while the BigQuery `DATE_DIFF` function outputs an integer. Therefore, a custom UDF function must be used to output the decimal point. For a detailed comparison between Netezza and BigQuery SQLs and functions, see Netezza to BigQuery SQL translation reference.

**Postmigration**

After the data migration is complete, you can begin to optimize your usage of Google Cloud to start solving business needs. This might include using Google Cloud’s explore and visualize tools to derive insights for business stakeholders, optimizing underperforming queries, or even developing a program to aid user adoption.
Connecting to BigQuery APIs over the internet

The following figure illustrates the options for ETL flows.

The following steps are observed:

1. In Google Cloud, a service account is created with BigQuery IAM permissions. The service account key is generated in JSON format and copied to the frontend server (for example, MicroStrategy).

2. The frontend reads the key and requests an OAuth token from Google APIs on HTTPS.

3. The frontend then sends BigQuery requests along with the token to BigQuery.

Optimizing for BigQuery

BigQuery’s standard SQL supports compliance with the SQL 2011 standard and has extensions that support querying nested and repeated data. Optimizing queries for BigQuery is critical in improving performance and response time and in reducing overall execution cost.
Replacing the Months_Between function in BigQuery with UDFs

Netezza treats the days in a month as 31, according to the official Netezza documentation. This custom UDF recreates the Netezza function with close accuracy.

```sql
CREATE TEMP FUNCTION months_between(date_1 DATE, date_2 DATE)
AS (CASE
    WHEN date_1 = date_2 THEN 0
    WHEN EXTRACT(DAY FROM DATE_ADD(date_1, INTERVAL 1 DAY)) = 1
    AND EXTRACT(DAY FROM DATE_ADD(date_2, INTERVAL 1 DAY)) = 1 THEN
date_diff(date_1, date_2, MONTH)
    WHEN EXTRACT(DAY FROM date_1) = 1 AND EXTRACT(DAY FROM
DATE_ADD(date_2, INTERVAL 1 DAY)) = 1 THEN date_diff(DATE_ADD(date_1, INTERVAL -1 DAY), date_2, MONTH) + 1/31
    ELSE date_diff(date_1, date_2, MONTH) - 1 + ((EXTRACT(DAY FROM date_1) + (31 - EXTRACT(DAY FROM date_2))) / 31)
END);
```

Netezza stored procedure migration

Netezza stored procedures used in ETL workloads for constructing fact tables need to be migrated to BigQuery-compatible SQL queries. Netezza uses the NZPLSQL scripting language to work with stored procedures. NZPLSQL is based on Postgres PL/pgSQL language. For more details, see the section “Procedural SQL statements” in Netezza to BigQuery SQL translation reference.

Custom UDF to emulate Netezza ASCII

This custom UDF corrects errors in columns.

```sql
CREATE TEMP FUNCTION ascii(X STRING)
AS (TO_CODE_POINTS(x)[ OFFSET (0)]);
```