Cloud Spanner supports simple data types such as integers, as well as more complex types such as ARRAY and STRUCT. This page provides an overview of each data type, including allowed values.

The maximum size of a column value is 10MiB, which applies to scalar and array types.

Type Name	Valid Column Type?	Valid Key Column Type?	Valid SQL Type?	Storage Size [1]
ARRAY (#array-type)	yes	no	yes	The sum of the size of its elements
BOOL (#boolean-type)	yes	yes	yes	1 byte
BYTES (#bytes-type)	yes	yes	yes	The number of bytes
DATE (#date-type)	yes	yes	yes	4 bytes
FLOAT64 (#floating-point-type)	yes	yes	yes	8 bytes
INT64 (#integer-type)	yes	yes	yes	8 bytes
STRING (#string-type)	yes	yes	yes	The number of bytes in its UTF-8 encoding
STRUCT (#struct-type)	no	no	yes	Not applicable
TIMESTAMP (#timestamp-type)	yes	yes	yes	12 bytes

[1] Each cell also has 8 bytes of storage overhead, in addition to the values listed.

When storing and querying data, it is helpful to keep the following data type properties in mind:

Property	Description	Applies To
Nullable	**NULL** is a valid value.	All data types.

Property	Description	Applies To
Orderable	Can be used in an **ORDER BY** clause.	All data types except for: • ARRAY • STRUCT
Groupable	Can generally appear in an expression following **GROUP BY**, **DISTINCT**, or **PARTITION BY**. However, **PARTITION BY** expressions cannot include the floating point types **FLOAT** and **DOUBLE**.	All data types except for: • ARRAY • STRUCT
Comparable	Values of the same type can be compared to each other.	All data types, with the following exceptions: ARRAY comparisons are not supported. Equality comparisons for STRUCTs are supported field by field, in field order. Field names are ignored. Less than and greater than comparisons are not supported. All types that support comparisons can be used in a **JOIN** condition. See JOIN Types (/spanner/docs/query-syntax#join_types) for an explanation of join conditions.

Numeric types include integer types and floating point types.

Integers are numeric values that do not have fractional components.

Name	Storage Size	Range
INT64	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Floating point values are approximate numeric values with fractional components.

Name	Storage Size	Description
FLOAT64	8 bytes	Double precision (approximate) decimal values.

When working with floating point numbers, there are special non-numeric values that need to be considered: `NaN` and `+/-inf`

t the floating point special values as **`Infinity`**, **`-Infinity`**, and **`NaN`** when using the Cloud Spanner REST and RPC
umented in TypeCode (REST) (/spanner/docs/reference/rest/v1/ResultSetMetadata#TypeCode) and TypeCode (RP
nner/docs/reference/rpc/google.spanner.v1#google.spanner.v1.TypeCode). The literals **`+inf`**, **`-inf`**, and **`nan`** are no
rted in the Cloud Spanner REST and RPC APIs.

Arithmetic operators provide standard IEEE-754 behavior for all finite input values that produce finite output and for all operations for which at least one input is non-finite.

Function calls and operators return an overflow error if the input is finite but the output would be non-finite. If the input contains non-finite values, the output can be non-finite. In general functions do not introduce `NaN`s or `+/-inf`. However, specific functions like `IEEE_DIVIDE` can return non-finite values on finite input. All such cases are noted explicitly in Mathematical functions (/spanner/docs/functions-and-operators#mathematical-functions).

Left Term	Operator	Right Term	Returns
Any value	**`+`**	**`NaN`**	**`NaN`**
1.0	**`+`**	**`+inf`**	**`+inf`**
1.0	**`+`**	**`-inf`**	**`-inf`**
`-inf`	**`+`**	**`+inf`**	**`NaN`**
Maximum **`FLOAT64`** value	**`+`**	Maximum **`FLOAT64`** value	Overflow error
Minimum **`FLOAT64`** value	**`/`**	2.0	0.0
1.0	**`/`**	**`0.0`**	"Divide by zero" error

Comparison operators provide standard IEEE-754 behavior for floating point input.

Left Term	Operator	Right Term	Returns
`NaN`	`=`	Any value	`FALSE`
`NaN`	`<`	Any value	`FALSE`
Any value	`<`	`NaN`	`FALSE`
-0.0	`=`	0.0	`TRUE`
-0.0	`<`	0.0	`FALSE`

Floating point values are sorted in this order, from least to greatest:

1. `NULL`
2. `NaN` — All `NaN` values are considered equal when sorting.
3. `-inf`
4. Negative numbers
5. 0 or -0 — All zero values are considered equal when sorting.
6. Positive numbers
7. `+inf`

Special floating point values are grouped this way, including both grouping done by a `GROUP BY` clause and grouping done by the `DISTINCT` keyword:

- `NULL`
- `NaN` — All `NaN` values are considered equal when grouping.
- `-inf`
- 0 or -0 — All zero values are considered equal when grouping.
- `+inf`

Name	Description
BOOL	Boolean values are represented by the keywords TRUE and FALSE (case insensitive).

Name	Description
STRING	Variable-length character (Unicode) data.

Input STRING values must be UTF-8 encoded and output STRING values will be UTF-8 encoded. Alternate encodings like CESU-8 and Modified UTF-8 are not treated as valid UTF-8.

All functions and operators that act on STRING values operate on Unicode characters rather than bytes. For example, functions like `SUBSTR` and `LENGTH` applied to STRING input count the number of characters, not bytes.

Each Unicode character has a numeric value called a code point assigned to it. Lower code points are assigned to lower characters. When characters are compared, the code points determine which characters are less than or greater than other characters.

Most functions on STRING are also defined on BYTES. The BYTES version operates on raw bytes rather than Unicode characters. STRING and BYTES are separate types that cannot be used interchangeably. There is no implicit casting in either direction. Explicit casting between STRING and BYTES does UTF-8 encoding and decoding. Casting BYTES to STRING returns an error if the bytes are not valid UTF-8.

Name	Description
BYTES	Variable-length binary data.

STRING and BYTES are separate types that cannot be used interchangeably. Most functions on STRING are also defined on BYTES. The BYTES version operates on raw bytes rather than Unicode characters. Casts between STRING and BYTES enforce that the bytes are encoded using UTF-8.

Name	Description	Range
`DATE`	Represents a logical calendar date.	0001-01-01 to 9999-12-31.

The DATE type represents a logical calendar date, independent of time zone. A DATE value does not represent a specific 24-hour time period. Rather, a given DATE value represents a different 24-hour period when interpreted in different time zones, and may represent a shorter or longer day during Daylight Savings Time transitions. To represent an absolute point in time, use a timestamp.

- `YYYY`: Four-digit year
- `[M]M`: One or two digit month
- `[D]D`: One or two digit day

Name	Description	Range
`TIMESTAMP`	Represents an absolute point in time, with nanosecond precision.	0001-01-01 00:00:00 to 9999-12-31 23:59:59.999999999 UTC.

A timestamp represents an absolute point in time, independent of any time zone or convention such as Daylight Savings Time.

TIMESTAMP provides nanosecond precision.

Follow the rules for encoding to and decoding from JSON values as described in TypeCode (RPC) (/spanner/docs/reference/rpc/google.spanner.v1#google.spanner.v1.TypeCode) and TypeCode (REST) (/spanner/docs/reference/rest/v1/ResultSetMetadata#TypeCode). In particular, the timestamp value must end with an uppercase literal "Z" to specify Zulu time (UTC-0).

For example:

Timestamp values must be expressed in Zulu time and cannot include a UTC offset. For example, the following timestamp is not supported:

Use the language-specific timestamp format.

- `YYYY`: Four-digit year
- `[M]M`: One or two digit month
- `[D]D`: One or two digit day
- `( |T)`: A space or a `T` separator
- `[H]H`: One or two digit hour (valid values from 00 to 23)
- `[M]M`: One or two digit minutes (valid values from 00 to 59)
- `[S]S`: One or two digit seconds (valid values from 00 to 59)
- `[.DDDDDDDDD]`: Up to nine fractional digits (nanosecond precision)

- `[time zone]`: String representing the time zone. See the time zones (#time-zones) section for details.

Time zones are used when parsing timestamps or formatting timestamps for display. The timestamp value itself does not store a specific time zone. A string-formatted timestamp may include a time zone. When a time zone is not explicitly specified, the default time zone, America/Los_Angeles, is used.

Time zones are represented by strings in one of these two canonical formats:

- Offset from Coordinated Universal Time (UTC), or the letter `Z` for UTC
- Time zone name from the tz database (http://www.iana.org/time-zones)

When using this format, no space is allowed between the time zone and the rest of the timestamp.

Time zone names are from the tz database (http://www.iana.org/time-zones). For a less comprehensive but simpler reference, see the List of tz database time zones (http://en.wikipedia.org/wiki/List_of_tz_database_time_zones) on Wikipedia.

When using a time zone name, a space is required between the name and the rest of the timestamp:

Note that not all time zone names are interchangeable even if they do happen to report the same time during a given part of the year. For example, `America/Los_Angeles` reports the same time as `UTC-7:00` during Daylight Savings Time, but reports the same time as `UTC-8:00` outside of Daylight Savings Time.

If a time zone is not specified, the default time zone value is used.

A timestamp is simply an offset from 1970-01-01 00:00:00 UTC, assuming there are exactly 60 seconds per minute. Leap seconds are not represented as part of a stored timestamp.

If the input contains values that use ":60" in the seconds field to represent a leap second, that leap second is not preserved when converting to a timestamp value. Instead that value is interpreted as a timestamp with ":00" in the seconds field of the following minute.

Leap seconds do not affect timestamp computations. All timestamp computations are done using Unix-style timestamps, which do not reflect leap seconds. Leap seconds are only observable through

functions that measure real-world time. In these functions, it is possible for a timestamp second to be skipped or repeated when there is a leap second.

Name	Description
`ARRAY`	Ordered list of zero or more elements of any non-ARRAY type.

An ARRAY is an ordered list of zero or more elements of non-ARRAY values. ARRAYs of ARRAYs are not allowed. Queries that would produce an ARRAY of ARRAYs will return an error. Instead a STRUCT must be inserted between the ARRAYs using the `SELECT AS STRUCT` construct.

An empty ARRAY and a `NULL` ARRAY are two distinct values. ARRAYs can contain `NULL` elements.

ARRAY types are declared using the angle brackets (`<` and `>`). The type of the elements of an ARRAY can be arbitrarily complex with the exception that an ARRAY cannot directly contain another ARRAY.

Type Declaration	Meaning
`ARRAY<INT64>`	Simple ARRAY of 64-bit integers.
`ARRAY<STRUCT<INT64, INT64>>`	An ARRAY of STRUCTs, each of which contains two 64-bit integers. **Note**: ARRAY of STRUCTs values can be constructed by SQL expressions, but are not supported as column types.

Type Declaration	Meaning
`ARRAY<ARRAY<INT64>>` (not supported)	This is an **invalid** type declaration which is included here just in case you came looking for how to create a multi-level ARRAY. ARRAYs cannot contain ARRAYs directly. Instead see the next example.
`ARRAY<STRUCT<ARRAY<INT64>>>`	An ARRAY of ARRAYS of 64-bit integers. Notice that there is a STRUCT between the two ARRAYs because ARRAYs cannot hold other ARRAYs directly.

Note: ARRAY of STRUCTs values can be constructed by SQL expressions, but are not supported as column types.

etails about using **`STRUCT`**s in **`SELECT`** statements (/spanner/docs/query-syntax#using-structs-with-select) an ueries (/spanner/docs/query-syntax#notes-about-subqueries) in the Query Syntax page.

Name	Description
`STRUCT`	Container of ordered fields each with a type (required) and field name (optional).

STRUCT types are declared using the angle brackets (`<` and `>`). The type of the elements of a STRUCT can be arbitrarily complex.

STRUCT values can be constructed by SQL expressions, but are not supported as column types.

Type Declaration	Meaning
`STRUCT<INT64>`	Simple STRUCT with a single unnamed 64-bit integer field.
`STRUCT<x STRUCT<y INT64, z INT64>>`	A STRUCT with a nested STRUCT named `x` inside it. The STRUCT `x` has two fields, `y` and `z`, both of which are 64-bit integers.
`STRUCT<inner_array ARRAY<INT64>>`	A STRUCT containing an ARRAY named `inner_array` that holds 64-bit integer elements.

The output type is an anonymous STRUCT type with anonymous fields with types matching the types of the input expressions. There must be at least two expressions specified. Otherwise this syntax is indistinguishable from an expression wrapped with parentheses.

Syntax	Output Type	Notes
`(x, x+y)`	`STRUCT<?, ?>`	If column names are used (unquoted strings), the STRUCT field data type is derived from the column data type. `x` and `y` are columns, so the data types of the STRUCT fields are derived from the column types and the output type of the addition operator.

This syntax can also be used with STRUCT comparison for comparison expressions using multi-part keys, e.g. in a `WHERE` clause:

Duplicate field names are allowed. Fields without names are considered anonymous fields and cannot be referenced by name. STRUCT values can be `NULL`, or can have `NULL` field values.

Syntax	Output Type
`STRUCT(1,2,3)`	`STRUCT<int64,int64,int64>`
`STRUCT()`	`STRUCT<>`
`STRUCT('abc')`	`STRUCT<string>`
`STRUCT(1, t.str_col)`	`STRUCT<int64, str_col string>`
`STRUCT(1 AS a, 'abc' AS b)`	`STRUCT<a int64, b string>`
`STRUCT(str_col AS abc)`	`STRUCT<abc string>`

Typed syntax allows constructing STRUCTs with an explicit STRUCT data type. The output type is exactly the `field_type` provided. The input expression is coerced to `field_type` if the two types are not the same, and an error is produced if the types are not compatible. `AS alias` is not allowed on the input expressions. The number of expressions must match the number of fields in the type, and the expression types must be coercible or literal-coercible to the field types.

Syntax	Output Type
`STRUCT<int64>(5)`	`STRUCT<int64>`

Syntax	Output Type
`STRUCT<date>("2011-05-05")`	`STRUCT<date>`
`STRUCT<x int64, y string>(1, t.str_col)`	`STRUCT<x int64, y string>`
`STRUCT<int64>(int_col)`	`STRUCT<int64>`
`STRUCT<x int64>(5 AS x)`	Error - Typed syntax does not allow `AS`

STRUCTs can be directly compared using equality operators:

- Equal (`=`)
- Not Equal (`!=` or `<>`)
- `[NOT] IN`

Notice, though, that these direct equality comparisons compare the fields of the STRUCT pairwise in ordinal order ignoring any field names. If instead you want to compare identically named fields of a STRUCT, you can compare the individual fields directly.