Dataflow (/dataflow/) is a managed service for transforming and enriching data. The Dataflow connector for Cloud Spanner lets you read data from and write data to Cloud Spanner in a Dataflow pipeline, optionally transforming or modifying the data. You can also create pipelines that transfer data between Cloud Spanner and other Google Cloud products.

The Dataflow connector is the recommended method for efficiently moving data into and out of Cloud Spanner in bulk. When working with individual databases, there are other methods you can use to import and export data:

- Use the Cloud Console to export (/spanner/docs/export) an individual database from Cloud Spanner to Cloud Storage in Avro (https://en.wikipedia.org/wiki/Apache_Avro) format.
- Use the Cloud Console to import (/spanner/docs/import) a database back into Cloud Spanner from files you exported to Cloud Storage.
- Use the REST API or `gcloud` command-line tool to run export (/dataflow/docs/templates/provided-templates#cloud_spanner_to_gcs_avro) or import (/dataflow/docs/templates/provided-templates#gcs_avro_to_cloud_spanner) jobs from Cloud Spanner to Cloud Storage and back (also using Avro format).

The Dataflow connector for Cloud Spanner is part of the Apache Beam Java SDK (https://beam.apache.org/documentation/sdks/javadoc/current/), and it provides an API for performing the above actions. See the Apache Beam programming guide (https://beam.apache.org/documentation/programming-guide/) for more information about some of the concepts discussed below, such as `PCollection` objects and transforms.

he Dataflow connector for Cloud Spanner only supports the Dataflow SDK 2.x for Java. For more information, see e notes: Dataflow SDK 2.x for Java (/dataflow/release-notes/release-notes-java-2).

To add the Google Cloud Dataflow connector to a Maven project, add the `beam-sdks-java-io-google-cloud-platform` Maven artifact to your `pom.xml` file as a dependency.

For example, assuming that your `pom.xml` file sets `beam.version` to the appropriate version number, you would add the following dependency:

To read from Cloud Spanner, apply the SpannerIO.read() (https://beam.apache.org/documentation/sdks/javadoc/current/org/apache/beam/sdk/io/gcp/spanner/SpannerIO.html#read--) transform. Configure the read using the methods in the `SpannerIO.Read` (https://beam.apache.org/documentation/sdks/javadoc/current/org/apache/beam/sdk/io/gcp/spanner/SpannerIO.Read.html) class. Applying the transform returns a `PCollection<Struct>` (https://beam.apache.org/documentation/sdks/javadoc/current/org/apache/beam/sdk/values/PCollection.html) , where each element in the collection represents an individual row returned by the read operation. You can read from Cloud Spanner with and without a specific SQL query, depending on your desired output.

Applying the `SpannerIO.read()` transform returns a consistent view of data by performing a strong read. Unless you specify otherwise, the result of the read is snapshotted at the time that you started the read. See reads (/spanner/docs/reads#read_types) for more information about the different types of reads Cloud Spanner can perform.

To read a specific set of data from Cloud Spanner, configure the transform using the `SpannerIO.Read.withQuery()` (https://beam.apache.org/documentation/sdks/javadoc/current/org/apache/beam/sdk/io/gcp/spanner/SpannerIO.Read.html#withQuery-java.lang.String-) method to specify a SQL query. For example:

java-docs-samples/blob/master/dataflow/spanner-io/src/main/java/com/example/dataflow/SpannerRead.java)

To read from a database without using a query, you can specify a table name and a list of columns, or you can read using an index. To read from selected columns, specify a table name and a list of columns when you construct your transform using `SpannerIO.read()` (https://beam.apache.org/documentation/sdks/javadoc/current/org/apache/beam/sdk/io/gcp/spanner/SpannerIO.Read.html#withQuery-java.lang.String-)
. For example:

a-docs-samples/blob/master/dataflow/spanner-io/src/main/java/com/example/dataflow/SpannerReadApi.java)

You can also read from the table using a specific set of keys as index values. To do so, build the read using an index (/spanner/docs/secondary-indexes) that contains the desired key values with the `SpannerIO.Read.withIndex()` (https://beam.apache.org/documentation/sdks/javadoc/current/org/apache/beam/sdk/io/gcp/spanner/SpannerIO.Read.html#withIndex-java.lang.String-)
method.

A transform is guaranteed to be executed on a consistent snapshot of data. To control the staleness (/spanner/docs/timestamp-bounds#timestamp_bound_types) of data, use the

`SpannerIO.Read.withTimestampBound()` (https://beam.apache.org/documentation/sdks/javadoc/current/org/apache/beam/sdk/io/gcp/spanner/SpannerIO.Read.html#withTimestampBound-com.google.cloud.spanner.TimestampBound-) method. See transactions (/spanner/docs/transactions) for more information.

If you want to read data from multiple tables at the same point in time to ensure data consistency, perform all of the reads in a single transaction. To do so, apply a `createTransaction()` (https://beam.apache.org/documentation/sdks/javadoc/current/org/apache/beam/sdk/io/gcp/spanner/SpannerIO.CreateTransaction.html) transform, creating a `PCollectionView<Transaction>` object which then creates a transaction. The resulting view can be passed to a read operation using `SpannerIO.Read.withTransaction()` (https://beam.apache.org/documentation/sdks/javadoc/current/org/apache/beam/sdk/io/gcp/spanner/SpannerIO.Read.html#withTransaction-org.apache.beam.sdk.values.PCollectionView-)
.

docs-samples/blob/master/dataflow/spanner-io/src/main/java/com/example/dataflow/TransactionalRead.java)

You can read data from all available tables in a Cloud Spanner database:

/a-docs-samples/blob/master/dataflow/spanner-io/src/main/java/com/example/dataflow/SpannerReadAll.java)

The Dataflow connector only supports Cloud Spanner SQL queries where the first operator in the query execution plan is a **Distributed Union** (/spanner/docs/query-execution-operators#distributed_union). If you attempt to read data from Cloud Spanner using a query and you get an exception stating that the query `does not have a DistributedUnion at the root`, follow the steps in Understanding how Cloud Spanner executes queries (/spanner/docs/sql-best-practices#how-execute-queries) to retrieve an execution plan for your query using the Cloud Console.

If your SQL query isn't supported, simplify it to a query that has a distributed union as the first operator in the query execution plan. Remove aggregate functions, as well as the operators `DISTINCT`, `GROUP BY`, and `ORDER`, as they are the operators that are most likely to prevent the query from working.

Use the `Mutation` (https://googleapis.dev/java/google-cloud-clients/latest/com/google/cloud/spanner/Mutation.html) class's `newInsertOrUpdateBuilder()` (https://googleapis.dev/java/google-cloud-clients/latest/com/google/cloud/spanner/Mutation.html#newInsertOrUpdateBuilder-java.lang.String-) method instead of the `newInsertBuilder()` (https://googleapis.dev/java/google-cloud-clients/latest/com/google/cloud/spanner/Mutation.html#newInsertBuilder-java.lang.String-) method unless absolutely necessary. Dataflow provides at-least-once guarantees, meaning that the mutation is likely to be written several times. As a result, insert mutations are likely to generate errors that cause the pipeline to fail. To prevent these errors, create insert-or-update mutations, which can be applied more than once.

You can write data to Cloud Spanner with the Dataflow connector by using a `SpannerIO.write()` (https://beam.apache.org/documentation/sdks/javadoc/current/index.html?org/apache/beam/sdk/io/gcp/spanner/SpannerIO.html) transform to execute a collection of input row mutations. The Dataflow connector groups mutations into batches for efficiency.

The following example shows how to apply a write transform to a `PCollection` of mutations:

java-docs-samples/blob/master/dataflow/spanner-io/src/main/java/com/example/dataflow/SpannerWrite.java)

If a transform unexpectedly stops before completion, mutations that have already been applied will not be rolled back.

The **`SpannerIO.write()`** transform does not guarantee that all of the mutations in the **`PCollection`** will be applie
cally, in a single transaction. If a small set of mutations must be applied atomically, see Applying groups of mutations
cally (#mutationgroup) below.

You can use the `MutationGroup` (https://beam.apache.org/documentation/sdks/javadoc/current/index.html?org/apache/beam/sdk/io/gcp/spanner/SpannerIO.html) class to ensure that a group of mutations are applied together atomically. Mutations in a `MutationGroup` are guaranteed to be submitted in the same transaction, but the transaction might be retried.

Mutation groups perform best when they are used to group together mutations that affect data stored close together in the key space. Because Cloud Spanner interleaves parent and child table data together in the parent table, that data is always close together in the key space. We recommend that you either structure your mutation group so that it contains one mutation that is applied to a parent table and additional mutations that are applied to child tables, or so that all of its mutations modify data that is close together in the key space. For more information about how Cloud Spanner stores parent and child table data, see Schema and data model (/spanner/docs/schema-and-data-model#parent-child_table_relationships). If you don't organize your mutation groups around the recommended table hierarchies, or if the data being accessed is not close together in the key space, Cloud Spanner might need to perform two-phase commits, which will result in slower performance. For more information, see Locality tradeoffs (/spanner/docs/whitepapers/optimizing-schema-design#tradeoffs_of_locality).

To use `MutationGroup`, build a `SpannerIO.write()` transform and call the `SpannerIO.Write.grouped()` (https://beam.apache.org/documentation/sdks/javadoc/current/org/apache/beam/sdk/io/gcp/spanner/SpannerIO.Write.html#grouped--) method, which returns a transform that you can then apply to a `PCollection` of `MutationGroup` objects.

When creating a `MutationGroup`, the first mutation listed becomes the primary mutation. If your mutation group affects both a parent and a child table, the primary mutation should be a mutation to the parent table. Otherwise, you can use any mutation as the primary mutation. The Dataflow connector uses the primary mutation to determine partition boundaries in order to efficiently batch mutations together.

For example, imagine that your application monitors behavior and flags problematic user behavior for review. For each flagged behavior, you want to update the `Users` table to block the user's access to your application, and you also need to record the incident in the `PendingReviews` table. To make sure that both of the tables are updated atomically, use a `MutationGroup`:

ocs-samples/blob/master/dataflow/spanner-io/src/main/java/com/example/dataflow/SpannerGroupWrite.java)

When creating a mutation group, the first mutation supplied as an argument becomes the primary mutation. In this case, the two tables are unrelated, so there is no clear primary mutation. We've selected `userMutation` as primary by placing it first. Applying the two mutations separately would be faster, but wouldn't guarantee atomicity, so the mutation group is the best choice in this situation.

- Learn more about designing an Apache Beam data pipeline (https://beam.apache.org/documentation/pipelines/design-your-pipeline/).
- Learn how to export (/spanner/docs/export) and import (/spanner/docs/import) Cloud Spanner databases in the Cloud Console using Dataflow.