A transaction in Cloud Spanner is a set of reads and writes that execute atomically at a single logical point in time across columns, rows, and tables in a database.

Cloud Spanner supports these transaction modes:

- **Locking read-write**. This type of transaction is the only transaction type that supports writing data into Cloud Spanner. These transactions rely on pessimistic locking and, if necessary, two-phase commit. Locking read-write transactions may abort, requiring the application to retry.
- **Read-only**. This transaction type provides guaranteed consistency across several reads, but does not allow writes. Read-only transactions can be configured to read at timestamps in the past. Read-only transactions do not need to be committed and do not take locks.
- **Partitioned DML**. This transaction type executes a Data Manipulation Language (DML) statement as Partitioned DML (/spanner/docs/dml-partitioned). Partitioned DML is designed for bulk updates and deletes, particularly periodic cleanup and backfilling.

This page describes the general properties and semantics of transactions in Cloud Spanner and introduces the read-write, read-only, and Partitioned DML transaction interfaces in Cloud Spanner.

Because read-only transactions can run at any replica, we recommend that you perform all of your transaction reads
nly transactions if possible. Use locking read-write transactions only under the scenarios described in the next sectio

Here are scenarios in which you should use a locking read-write transaction:

- If you do a write that depends on the result of one or more reads, you should do that write and the read(s) in the same read-write transaction.
    - Example: double the balance of bank account A. The read of A's balance should be in the same transaction as the write to replace the balance with the doubled value.

- If you do one or more writes that need to be committed atomically, you should do those writes in the same read-write transaction.
    - Example: transfer $200 from account A to account B. Both of the writes (one to decrease A by $200 and one to increase B by $200) and the reads of initial account balances should be in the same transaction.

- If you **might** do one or more writes, depending on the results of one or more reads, you should do those writes and reads in the same read-write transaction, even if the write(s) don't end up executing.
    - Example: transfer $200 from bank account A to bank account B if A's current balance is greater than $500. Your transaction should contain a read of A's balance and a conditional statement that contains the writes.

Here is a scenario in which you should **not** use a locking read-write transaction:

- If you are only doing reads, and you can express your read using a single read method (/spanner/docs/reads#single_read_methods), you should use that single read method or a read-only transaction. Single reads do not lock, unlike read-write transactions.

A read-write transaction in Cloud Spanner executes a set of reads and writes atomically at a single logical point in time. Furthermore, the timestamp at which read-write transactions execute matches wall clock time, and the serialization order matches the timestamp order.

Why use a read-write transaction? Read-write transactions provide the ACID properties of relational databases (In fact, Cloud Spanner read-write transactions offer even stronger guarantees than traditional ACID; see the Semantics (#rw_transaction_semantics) section below.).

Here are the isolation properties you get for a read-write transaction that contains a series of reads and writes:

- All reads within that transaction return data from the same timestamp.
- If a transaction successfully commits, then no other writer modified the data that was read in the transaction after it was read.

- These properties hold even for reads that returned no rows, and the gaps between rows returned by range reads: row non-existence counts as data.
- All writes within that transaction are committed at the same timestamp.
- All writes within that transaction are only visible after the transaction commits.

The effect is that all reads and writes appear to have occurred at a single point in time, both from the perspective of the transaction itself and from the perspective of other readers and writers to the Cloud Spanner database. In other words, the reads and the writes end up occurring at the same timestamp (see an illustration of this in the Serializability and external consistency (#serializability_and_external_consistency) section below).

The guarantees for a read-write transaction that only reads are similar: all reads within that transaction return data from the same timestamp, even for row non-existence. One difference is that if you read data, and later commit the read-write transaction without any writes, there is no guarantee that the data did not change in the database after the read and before the commit. If you want to know whether data has changed since you read it last, the best approach is to read it again (either in a read-write transaction, or using a strong read.) Also, for efficiency, if you know in advance that you'll only be reading and not writing, you should use a read-only transaction (#read-only_transactions) instead of a read-write transaction.

In addition to the Isolation property, Cloud Spanner provides Atomicity (if any of the writes in the transaction commit, they all commit), Consistency (the database remains in a consistent state after the transaction) and Durability (committed data stays committed.)

Because of these properties, as an application developer, you can focus on the correctness of each transaction on its own, without worrying about how to protect its execution from other transactions that might execute at the same time.

Cloud Spanner provides an interface for executing a body of work in the context of a read-write transaction, with retries for transaction aborts. Here's a bit of context to explain this point: a Cloud Spanner transaction may have to be tried multiple times before it commits. For example, if two

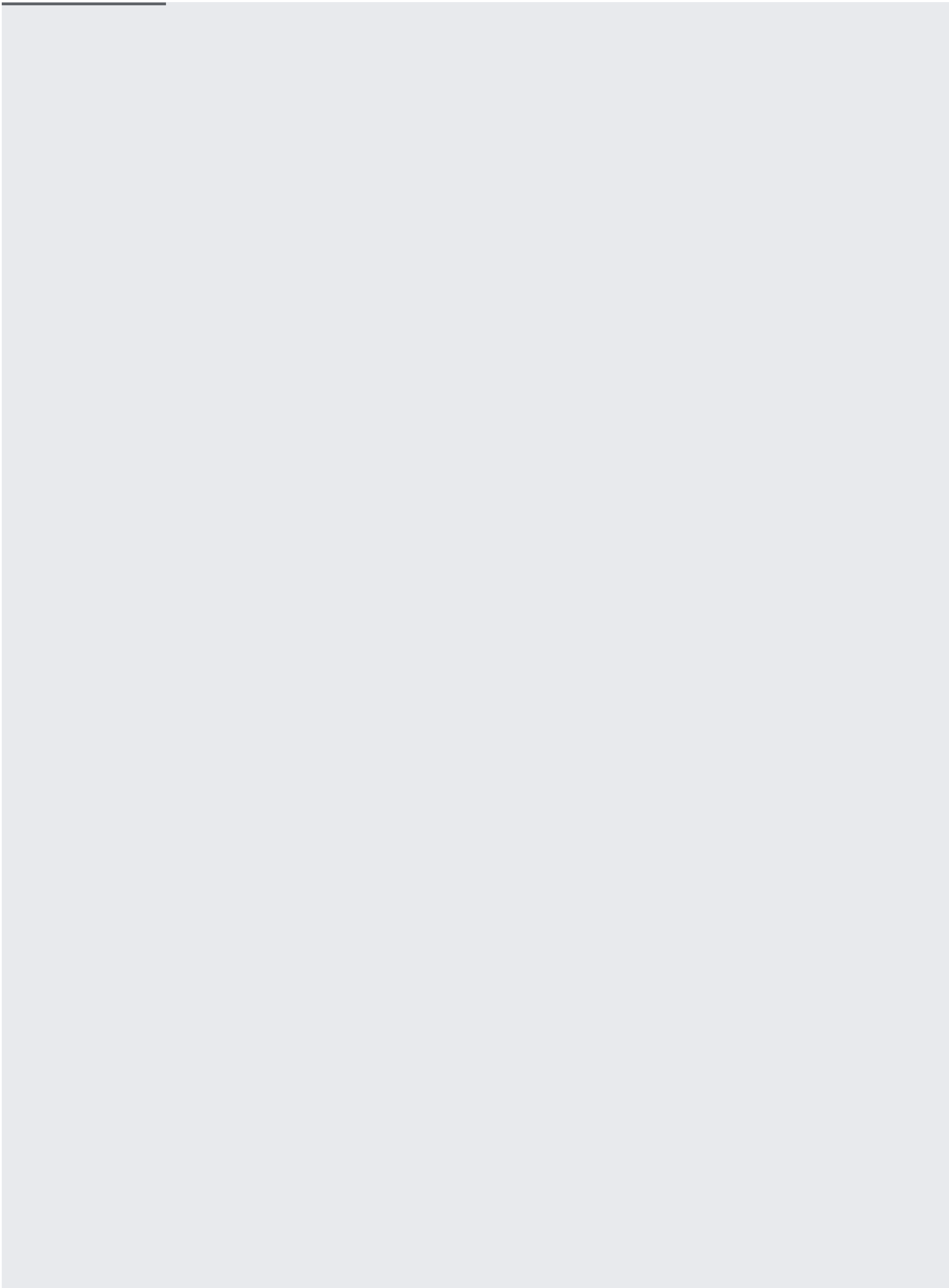
transactions attempt to work on data at the same time in a way that might cause deadlock, Cloud Spanner aborts one of them so that the other transaction can make progress. (More rarely, transient events within Cloud Spanner may result in some transactions aborting.) Since transactions are atomic, an aborted transaction has no visible effect on the database. Therefore, transactions should be executed by retrying them until they succeed.

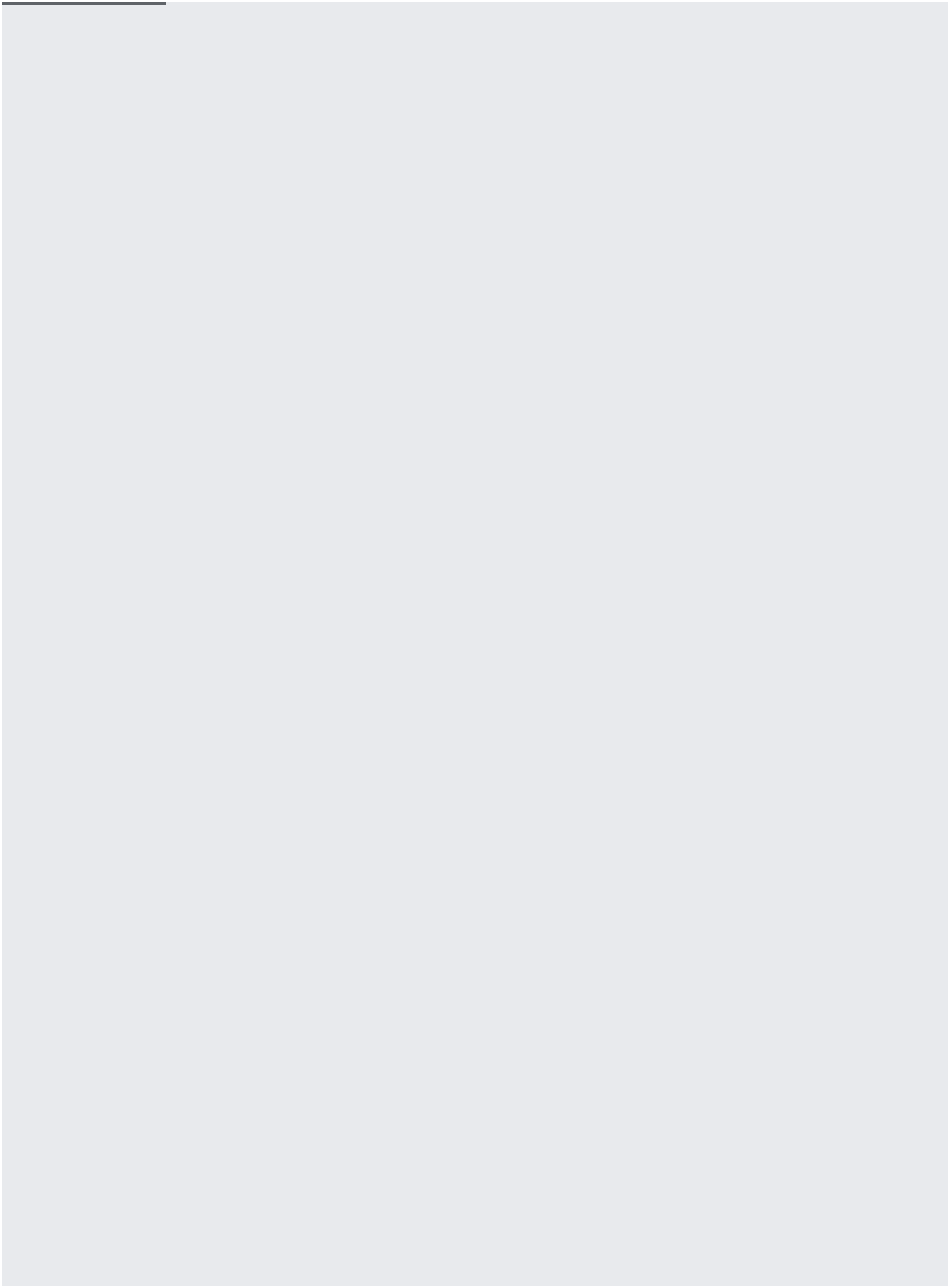
When you use a transaction in the Cloud Spanner API, you define the body of a transaction (i.e., the reads and writes to perform on one or more tables in a database) in the form of a function object. Under the hood, Cloud Spanner runs the function repeatedly until the transaction commits or a non-retryable error is encountered.

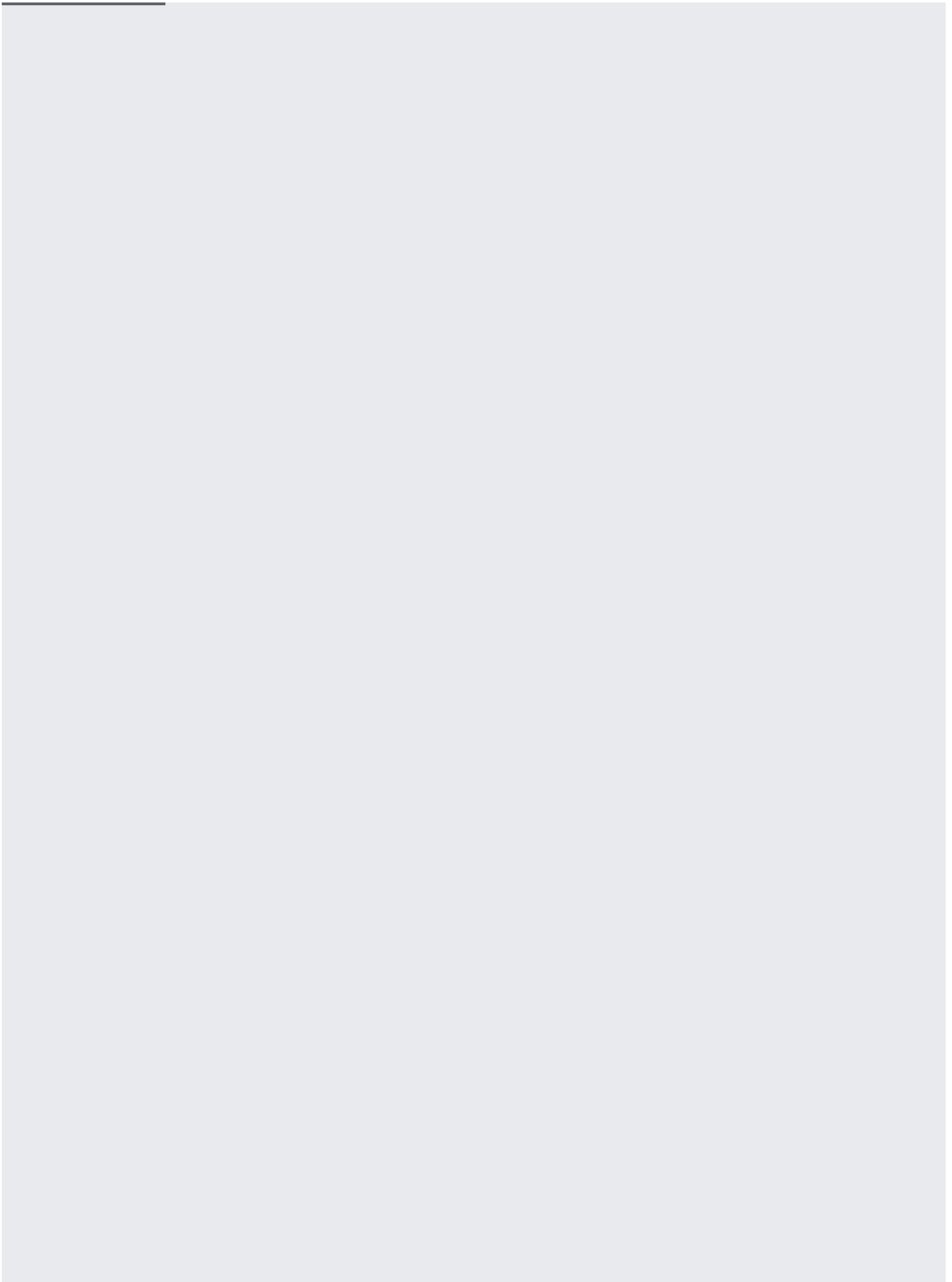
Assume you added a `MarketingBudget` column to the `Albums` table (/spanner/docs/schema-and-data-model#creating_multiple_tables) shown in the Schema and Data Model page:

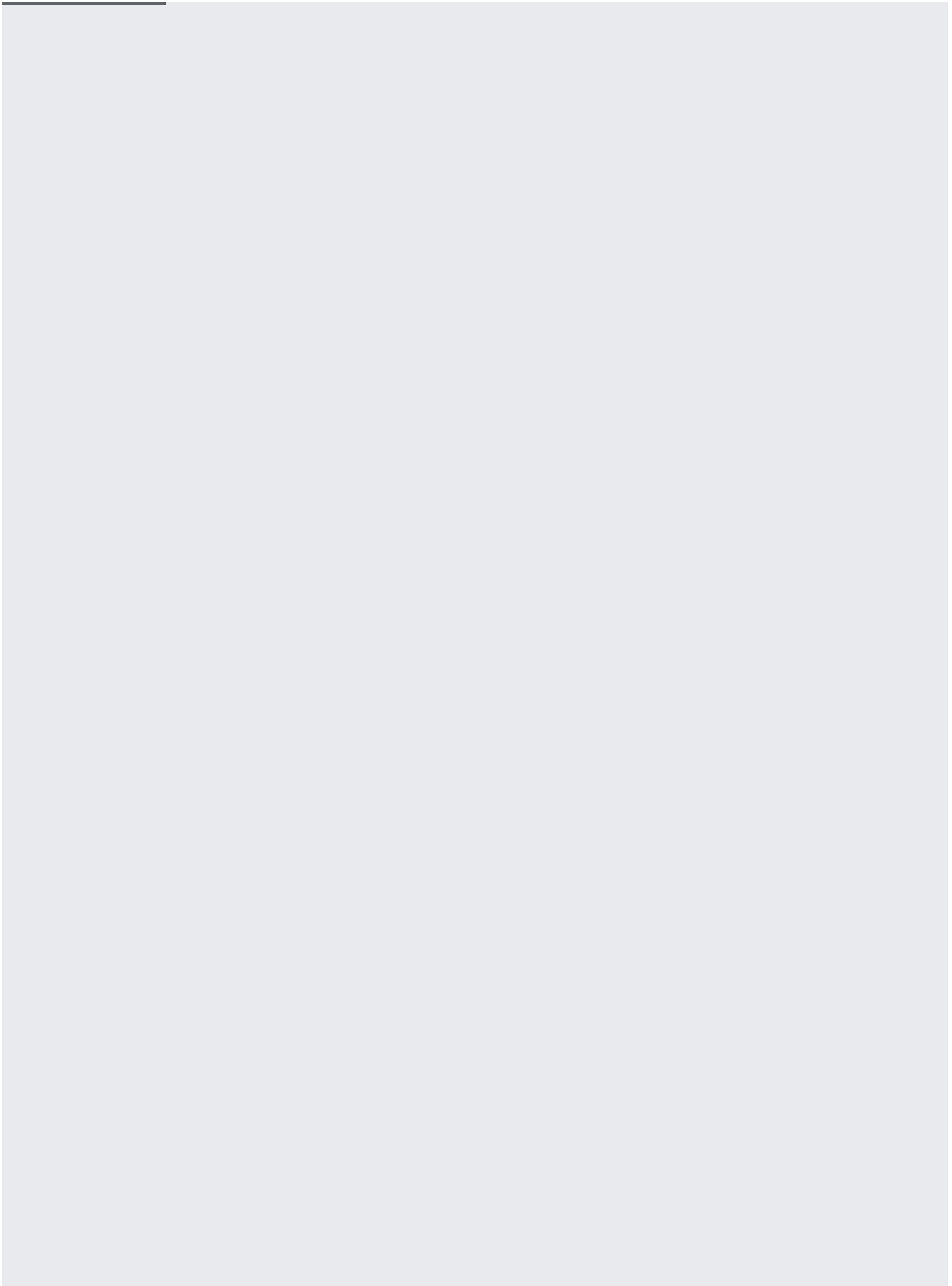
Your marketing department decides to do a marketing push for the album keyed by `Albums (1, 1)` and has asked you to move $200,000 from the budget of `Albums (2, 2)`, but only if the money is available in that album's budget. You should use a locking read-write transaction for this operation, because the transaction might do writes depending on the result of a read.

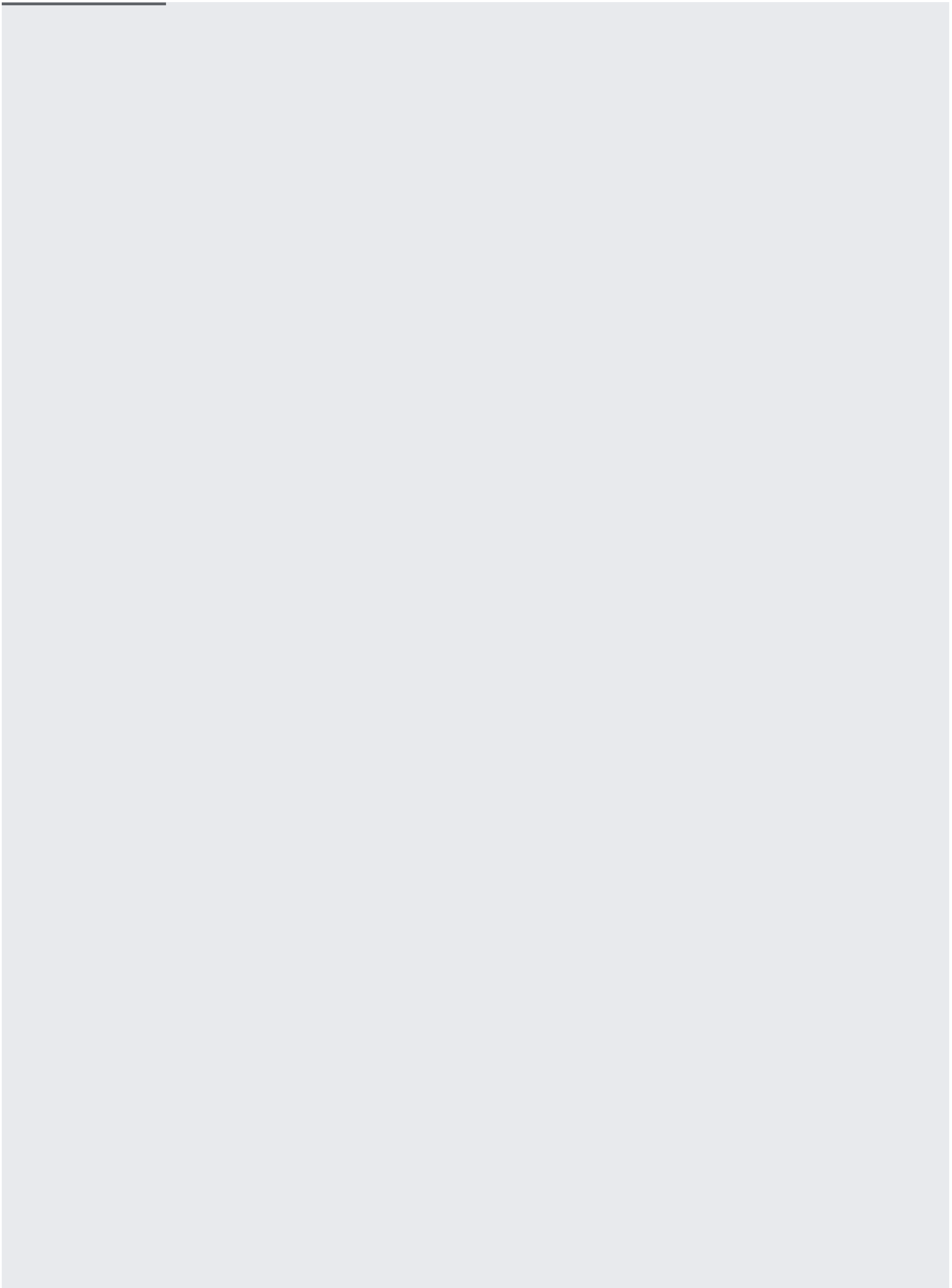
The following shows how to execute a read-write transaction:

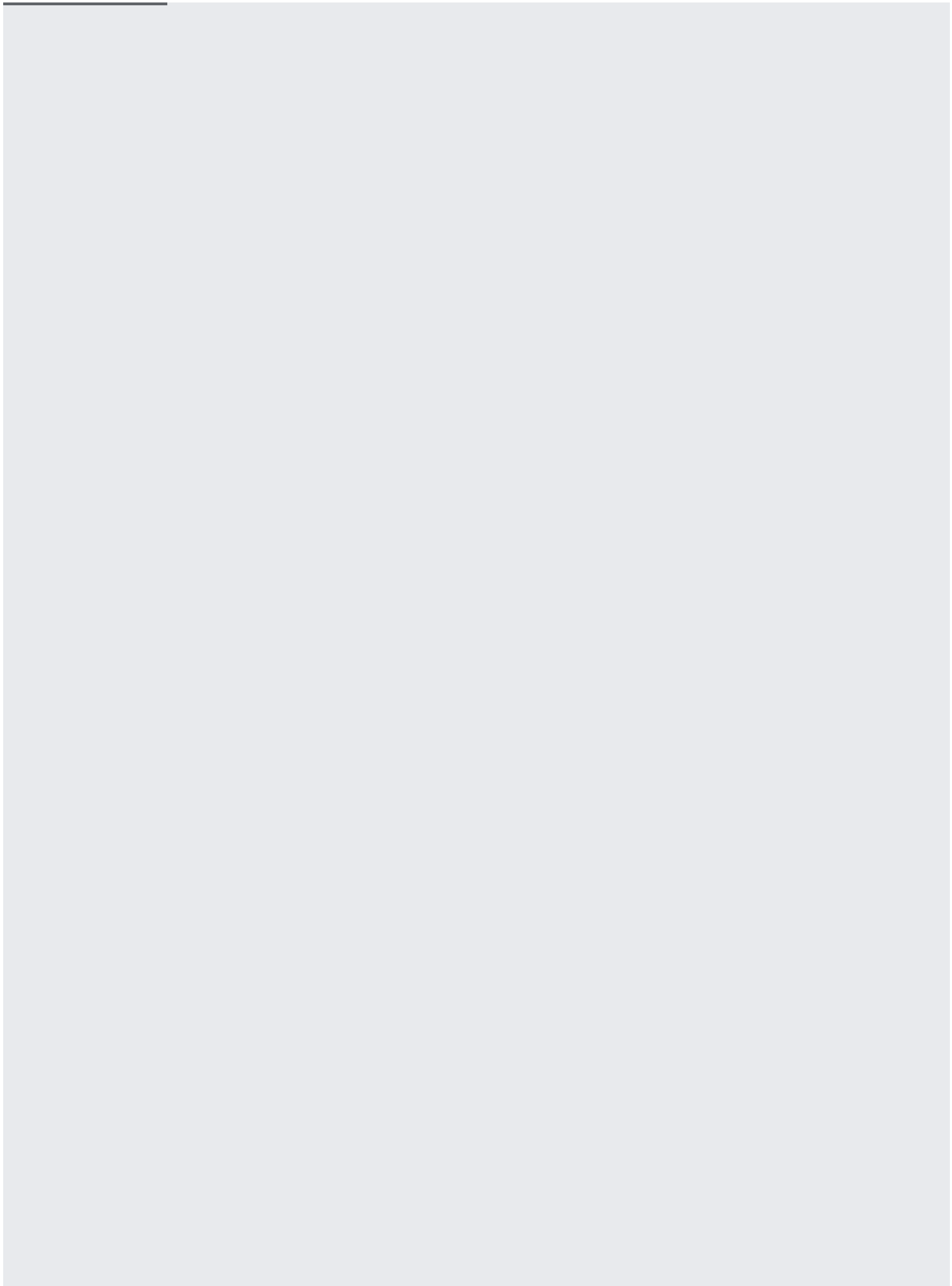


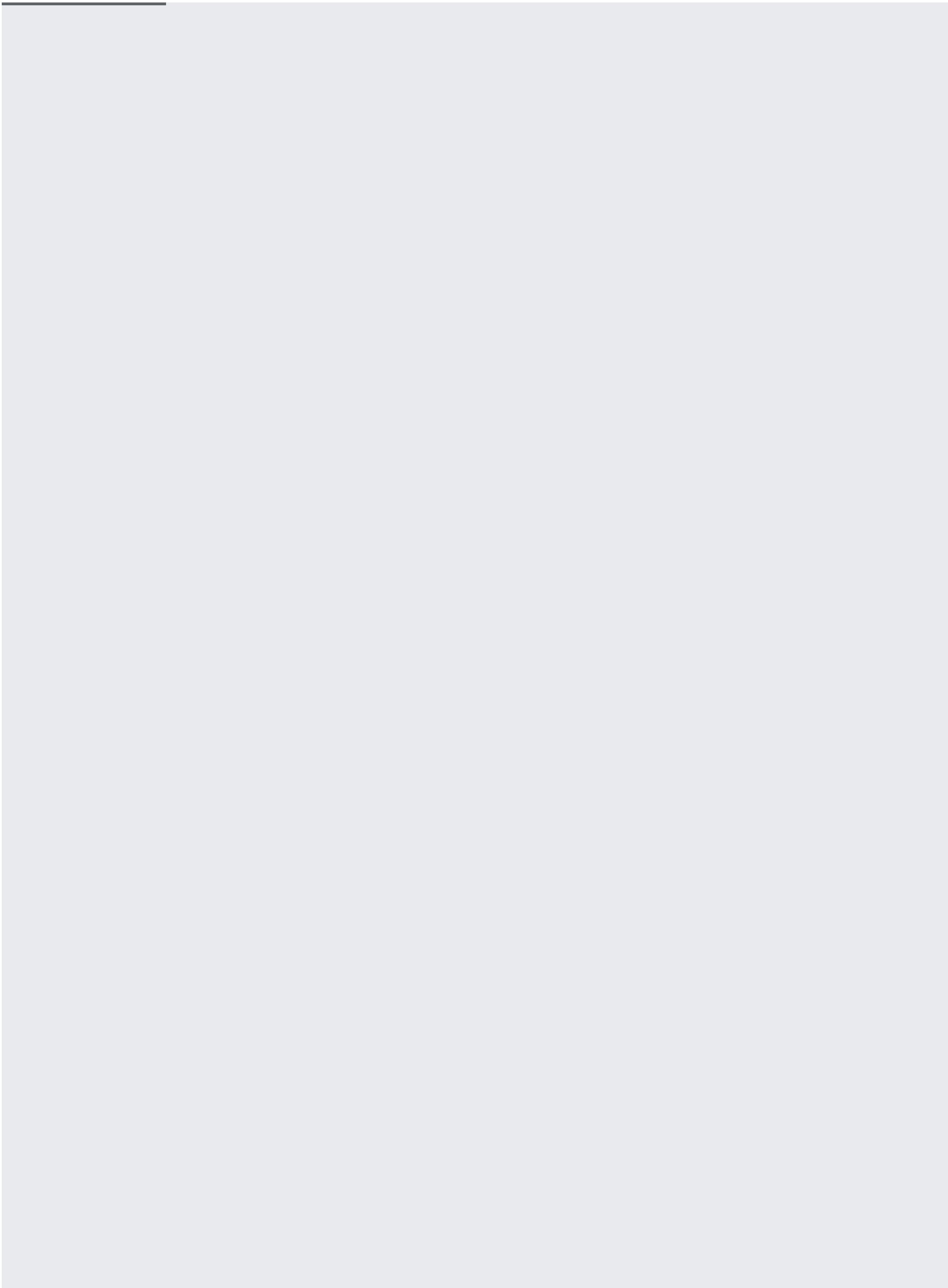


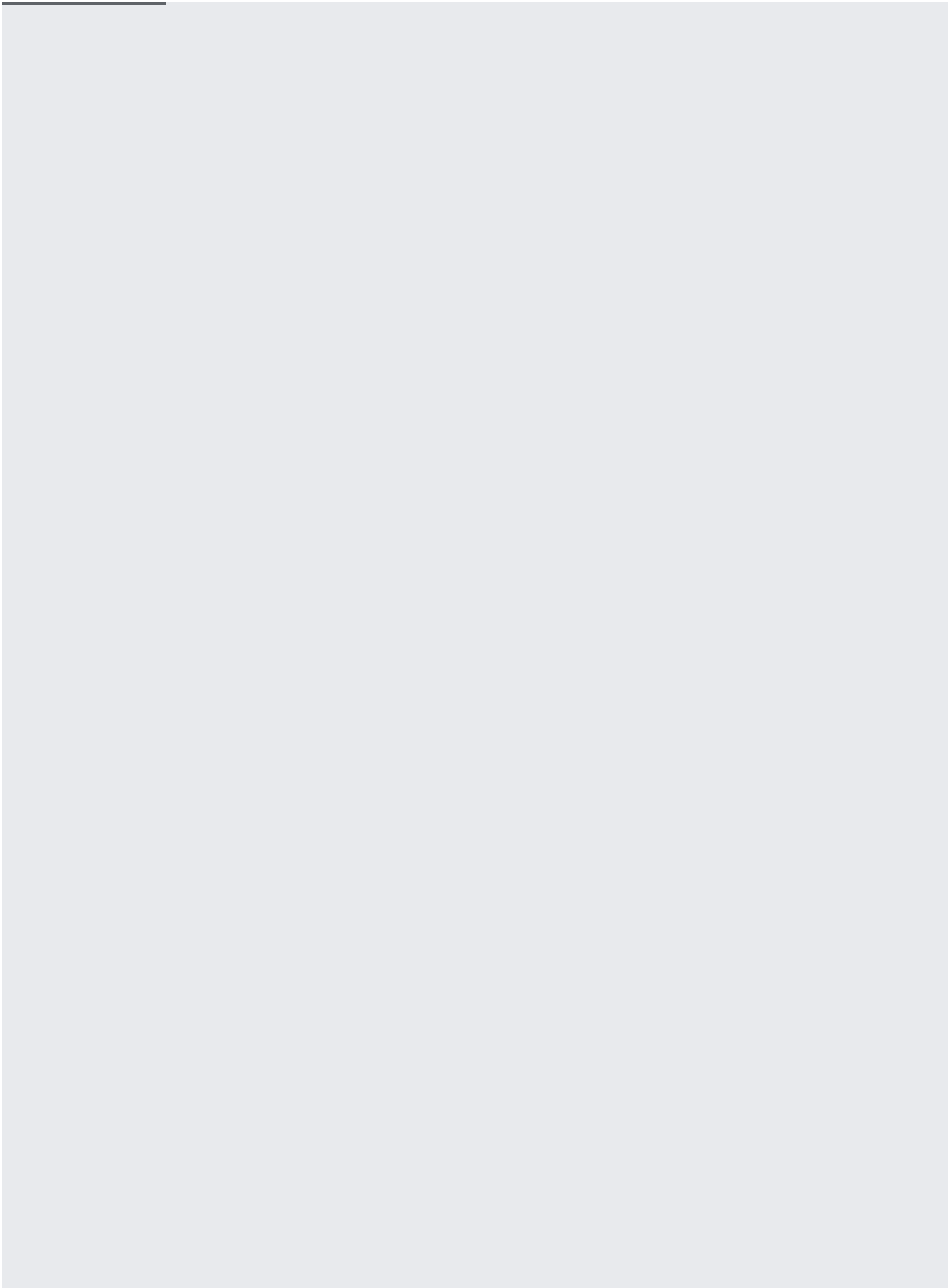


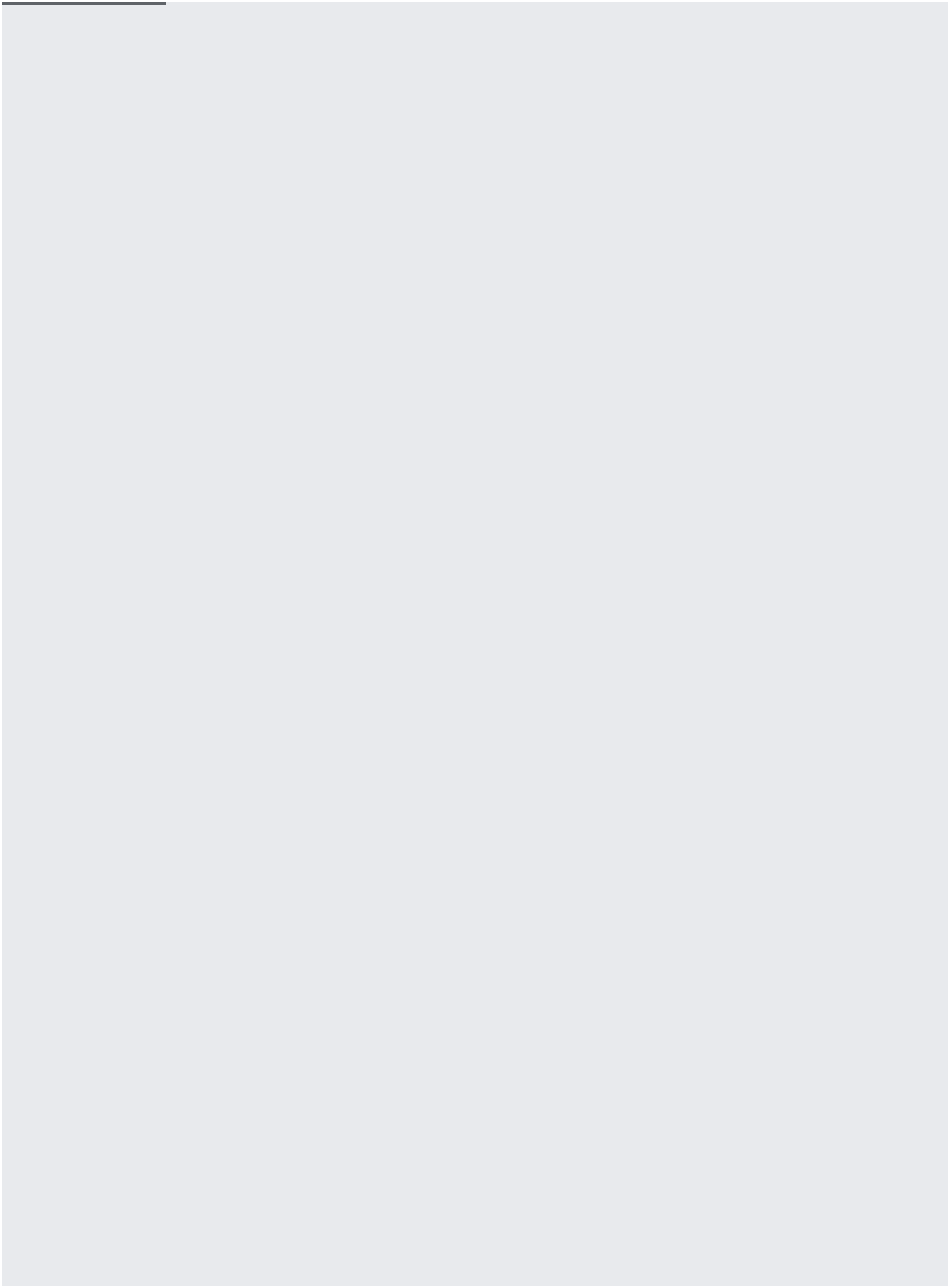


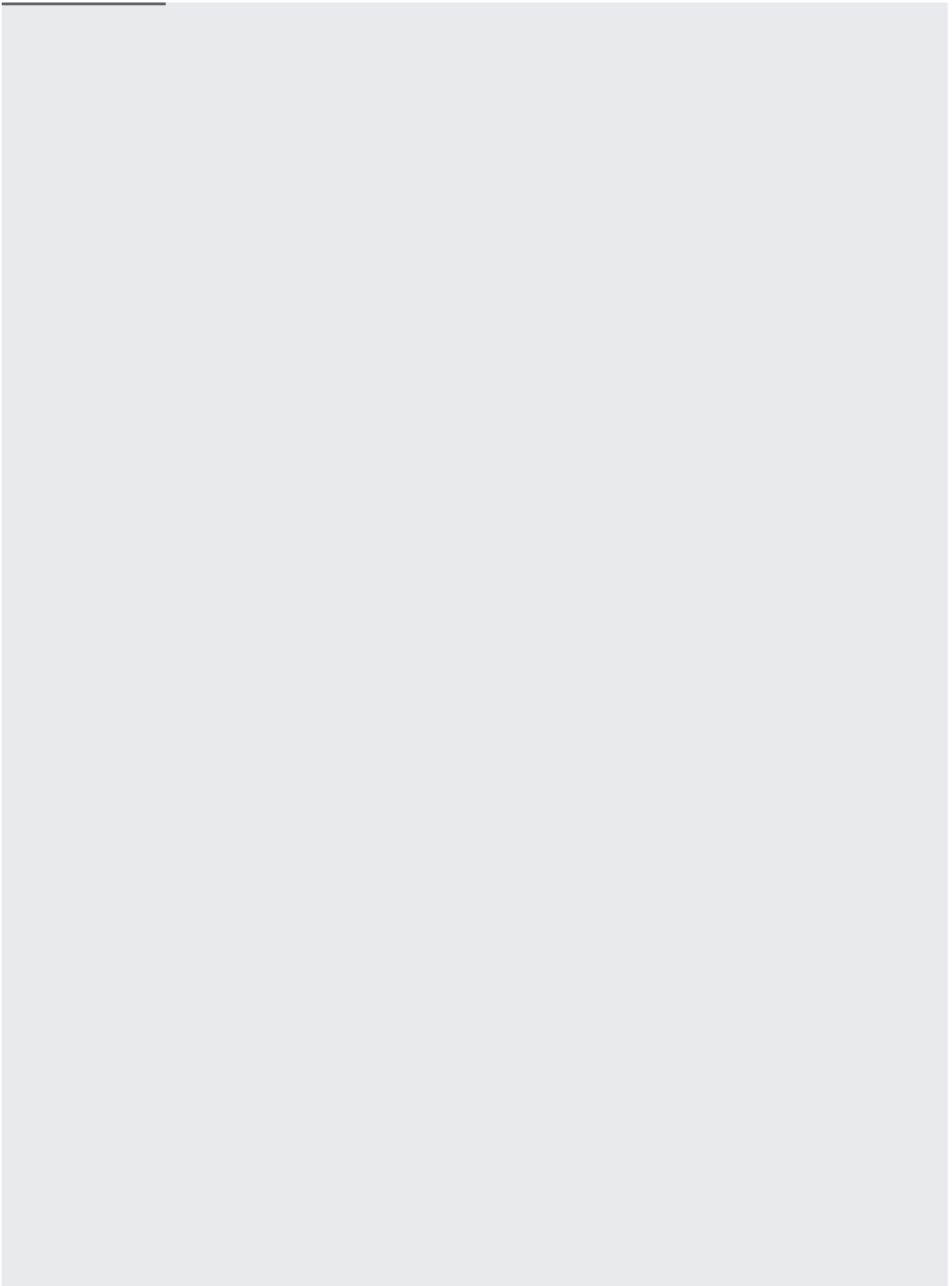






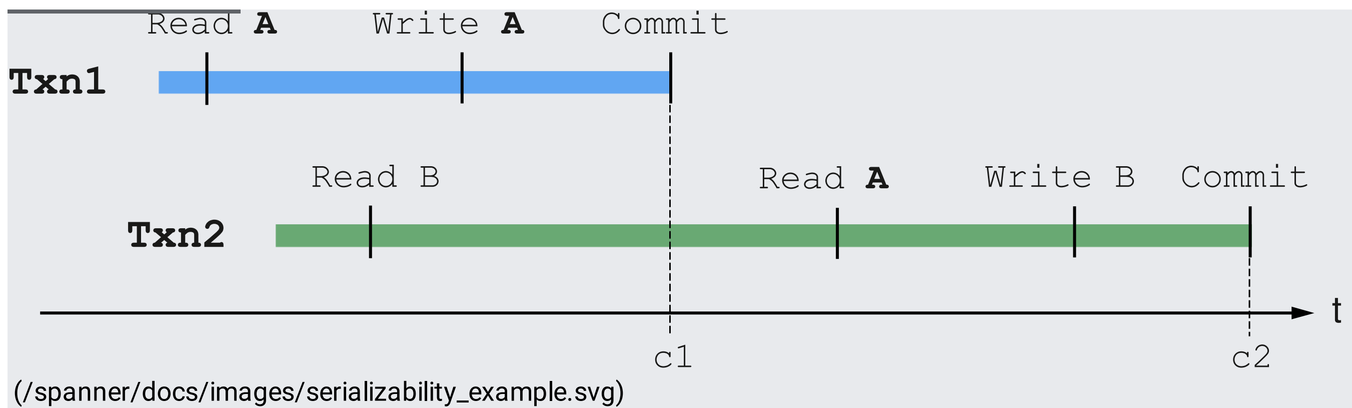






Cloud Spanner provides 'serializability', which means that all transactions appear as if they executed in a serial order, even if some of the reads, writes, and other operations of distinct transactions actually occurred in parallel. Cloud Spanner assigns commit timestamps that reflect the order of committed transactions to implement this property. In fact, Cloud Spanner offers a stronger guarantee than serializability called external consistency (/spanner/docs/true-time-external-consistency): transactions commit in an order that is reflected in their commit timestamps, and these commit timestamps reflect real time so you can compare them to your watch. Reads in a transaction see everything that has been committed before the transaction commits, and writes are seen by everything that starts after the transaction is committed.

For example, consider the execution of two transactions as illustrated in the diagram below:



(/spanner/docs/images/serializability_example.svg)

Transaction `Txn1` in blue reads some data `A`, buffers a write to `A`, then successfully commits. Transaction `Txn2` in green starts after `Txn1`, reads some data `B`, then reads the data `A`. **Since `Txn2` reads the value of `A` after `Txn1` committed its write to `A`, `Txn2` sees the effect of `Txn1`'s write to `A`, even though `Txn2` started before `Txn1` completed**.

Even though there is some overlap in time in which `Txn1` and `Txn2` are both executing, their commit timestamps `c1` and `c2` respect a linear transaction order, which means that all effects of the reads and writes of `Txn1` appear to have occurred at a single point of time (`c1`), and all effects of the reads and writes of `Txn2` appear to have occurred at a single point of time (`c2`). Furthermore, `c1 < c2` (which is guaranteed because both `Txn1` and `Txn2` committed writes; this is true even if the writes happened on different machines), which respects the order of `Txn1` happening before `Txn2`. (However, if `Txn2` only did reads in the transaction, then `c1 <= c2`).

Reads observe a prefix of the commit history; if a read sees the effect of `Txn2`, it also sees the effect of `Txn1`. All transactions that commit successfully have this property.

action reads don't see the same transaction's writes. It's important to note that reads do not see the w
re buffered in the same transaction. Writes are buffered until the end of the transaction and aren't visi
eads until the transaction is committed. This is because when you buffer mutations, they're stored loca
client and aren't sent to the server until commit time. Reads however, are sent directly to the server.

If a call to run a transaction fails, the read and write guarantees you have depend on what error the underlying commit call failed with.

For example, an error such as "Row Not Found" or "Row Already Exists" means that writing the buffered mutations encountered some error, e.g. a row that the client is trying to update doesn't exist. In that case, the reads are guaranteed consistent, the writes are not applied, and the non-existence of the row is guaranteed to be consistent with the reads as well.

Cloud Spanner might try to execute the body of the transaction multiple times under the hood. If an execution attemp
he error it returns indicates the condition(s) that occurred and thus, which guarantees you get. However, be aware th
de effects of your transaction body (such as changes to non-Cloud Spanner state in your program or other systems)
multiple times if Cloud Spanner has to retry your transaction multiple times.

Asynchronous read operations may be cancelled any time by the user (e.g., when a higher level operation is cancelled or you decide to stop a read based on the initial results received from the read) without affecting any other existing operations within the transaction.

However, even if you have attempted to cancel the read, Cloud Spanner does not guarantee that the read is actually cancelled. After you request the cancellation of a read, that read can still successfully complete or fail with some other reason (e.g. Abort). Furthermore, that cancelled read might actually return some results to you, and those possibly incomplete results will be validated as part of the transaction Commit.

Note that unlike reads, cancelling a transaction Commit operation will result in aborting the transaction (unless the transaction has already Committed or failed with another reason).

Cloud Spanner allows multiple clients to simultaneously interact with the same database. In order to ensure the consistency of multiple concurrent transactions, Cloud Spanner uses a combination of shared locks and exclusive locks to control access to the data. When you perform a read as part of a transaction, Cloud Spanner acquires shared read locks, which allows other reads to still access the data until your transaction is ready to commit. When your transaction is committing and writes are being applied, the transaction attempts to upgrade to an exclusive lock. It blocks new shared read locks on the data, waits for existing shared read locks to clear, then places an exclusive lock for exclusive access to the data.

Notes about locks:

- Locks are taken at the granularity of row-and-column. If transaction T1 has locked column "A" of row "foo", and transaction T2 wants to write column "B" of row "foo" then there is no conflict.
- Writes to a data item that don't also read the data being written (aka "blind writes") don't conflict with other blind writers of the same item (the commit timestamp of each write determines the

order in which it is applied to the database). A consequence of this is that Cloud Spanner only needs to upgrade to an exclusive lock if you have read the data you are writing. Otherwise Cloud Spanner uses a shared lock called a writer shared lock.

Cloud Spanner detects when multiple transactions might be deadlocked, and forces all but one of the transactions to abort. For example, consider the following scenario: transaction `Txn1` holds a lock on record `A` and is waiting for a lock on record `B`, and `Txn2` holds a lock on record `B` and is waiting for a lock on record `A`. The only way to make progress in this situation is to abort one of the transactions so it releases its lock, allowing the other transaction to proceed.

Cloud Spanner uses the standard "wound-wait" algorithm to handle deadlock detection. Under the hood, Cloud Spanner keeps track of the age of each transaction that requests conflicting locks. It also allows older transactions to abort younger transactions (where "older" means that the transaction's earliest read, query, or commit happened sooner).

By giving priority to older transactions, Cloud Spanner ensures that every transaction has a chance to acquire locks eventually, after it becomes old enough to have higher priority than other transactions. For example, a transaction that acquires a reader shared lock can be aborted by an older transaction that needs a writer shared lock.

Cloud Spanner can perform **distributed transactions**, which are transactions that touch many parts of the database, even if they are on different servers. This power comes at a performance cost compared to single-site transactions.

What types of transactions might be distributed? Under the hood, Cloud Spanner can divide responsibility for rows in the database across many servers. A row and the corresponding rows in interleaved tables are usually served by the same server, as are two rows in the same table with nearby keys. Cloud Spanner can perform transactions across rows on different servers; however, as a rule of thumb, transactions that affect many co-located rows are faster and cheaper than transactions that affect many rows scattered throughout the database, or throughout a large table.

The most efficient transactions in Cloud Spanner include only the reads and writes that should be applied atomically. Transactions are fastest when all reads and writes access data in the same part of the key space.

In addition to locking read-write transactions, Cloud Spanner offers read-only transactions.

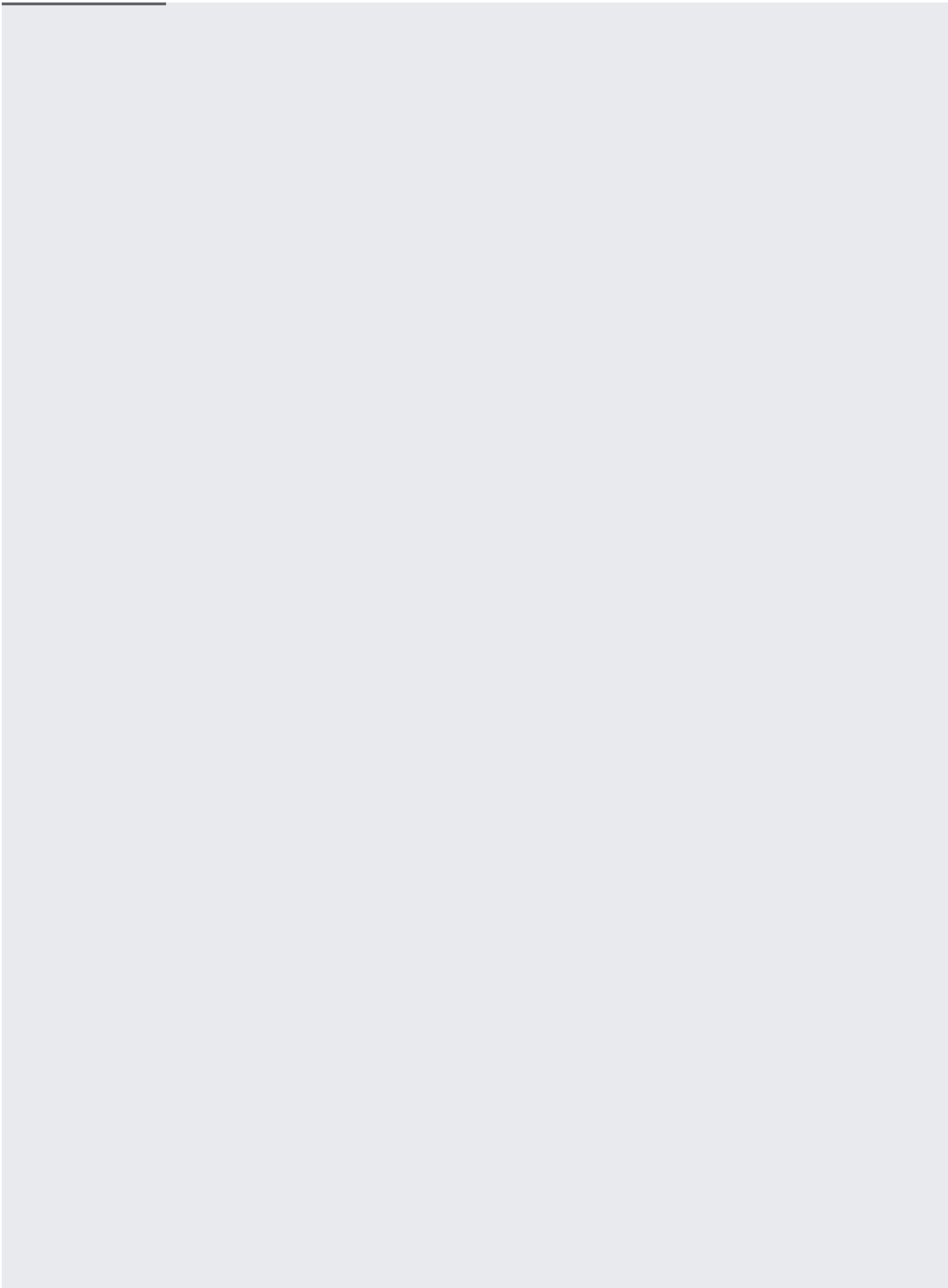
Use a read-only transaction when you need to execute more than one read at the same timestamp. If you can express your read using one of Cloud Spanner's single read methods (/spanner/docs/reads#single_read_methods), you should use that single read method instead. The performance of using such a single read call should be comparable to the performance of a single read done in a read-only transaction.

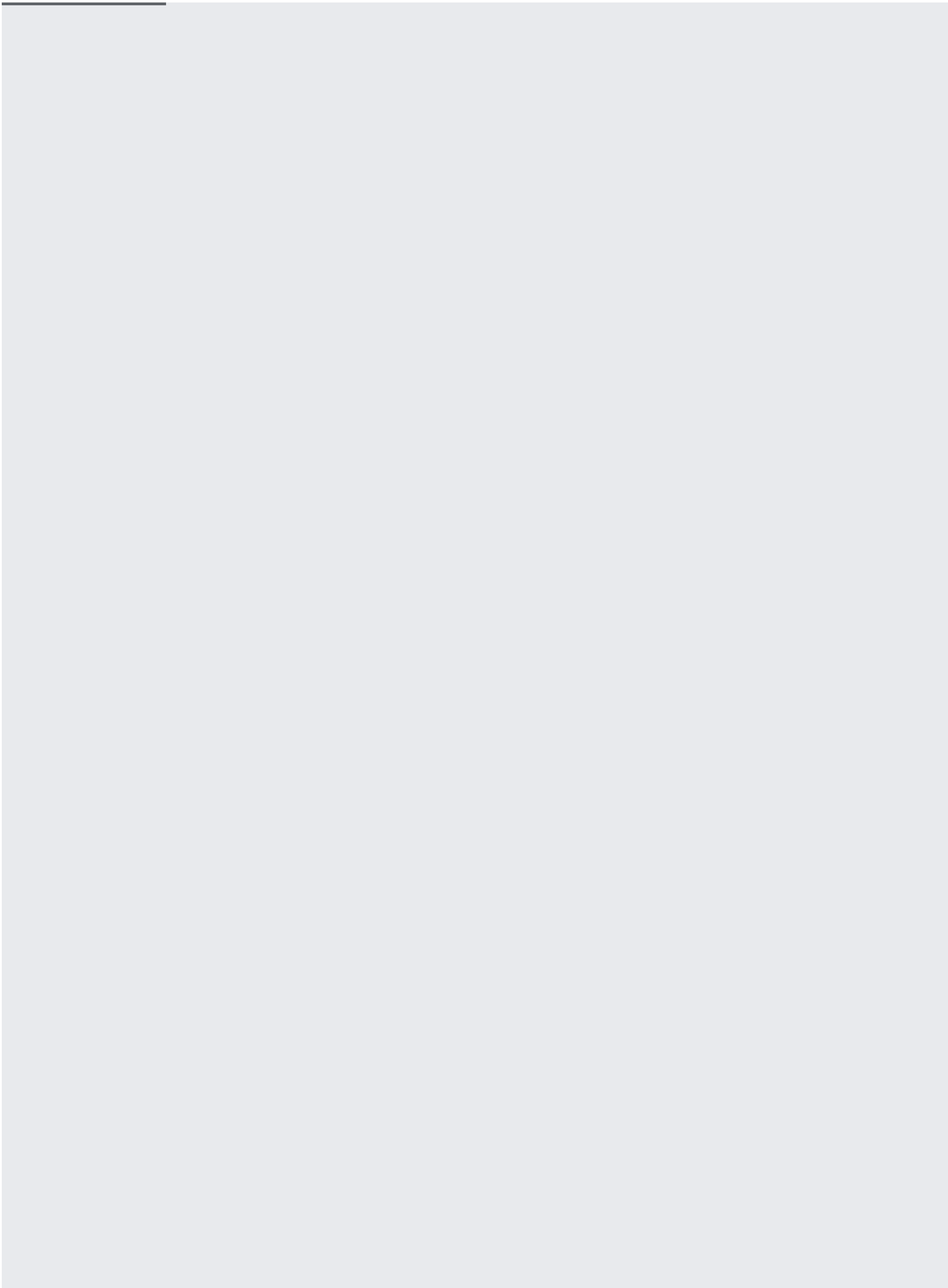
Because read-only transactions don't write, they don't hold locks and they don't block other transactions. Read-only transactions observe a consistent prefix of the transaction commit history, so your application always gets consistent data.

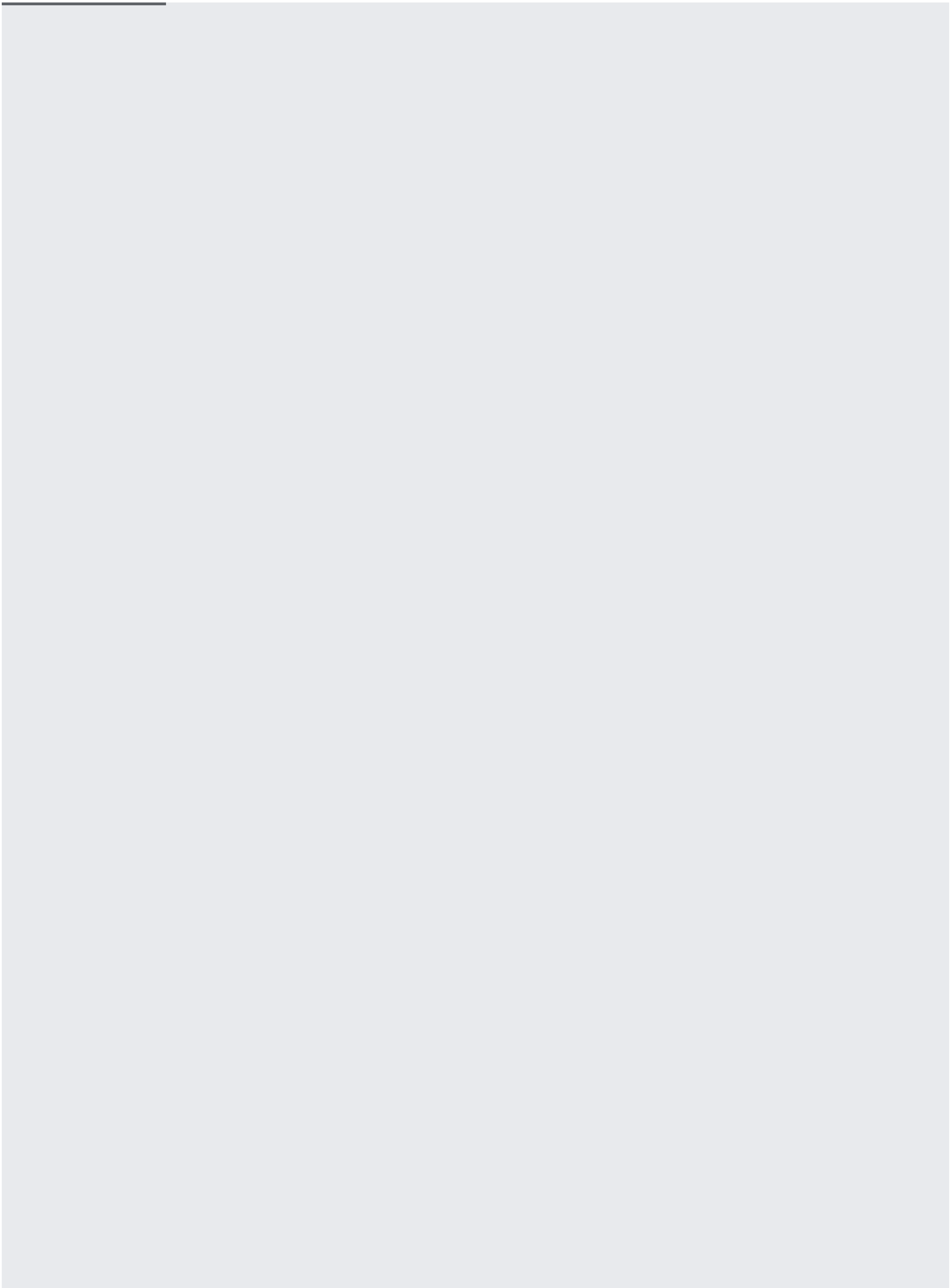
A Cloud Spanner read-only transaction executes a set of reads at a single logical point in time, both from the perspective of the read-only transaction itself and from the perspective of other readers and writers to the Cloud Spanner database. This means that read-only transactions always observe a consistent state of the database at a chosen point in the transaction history.

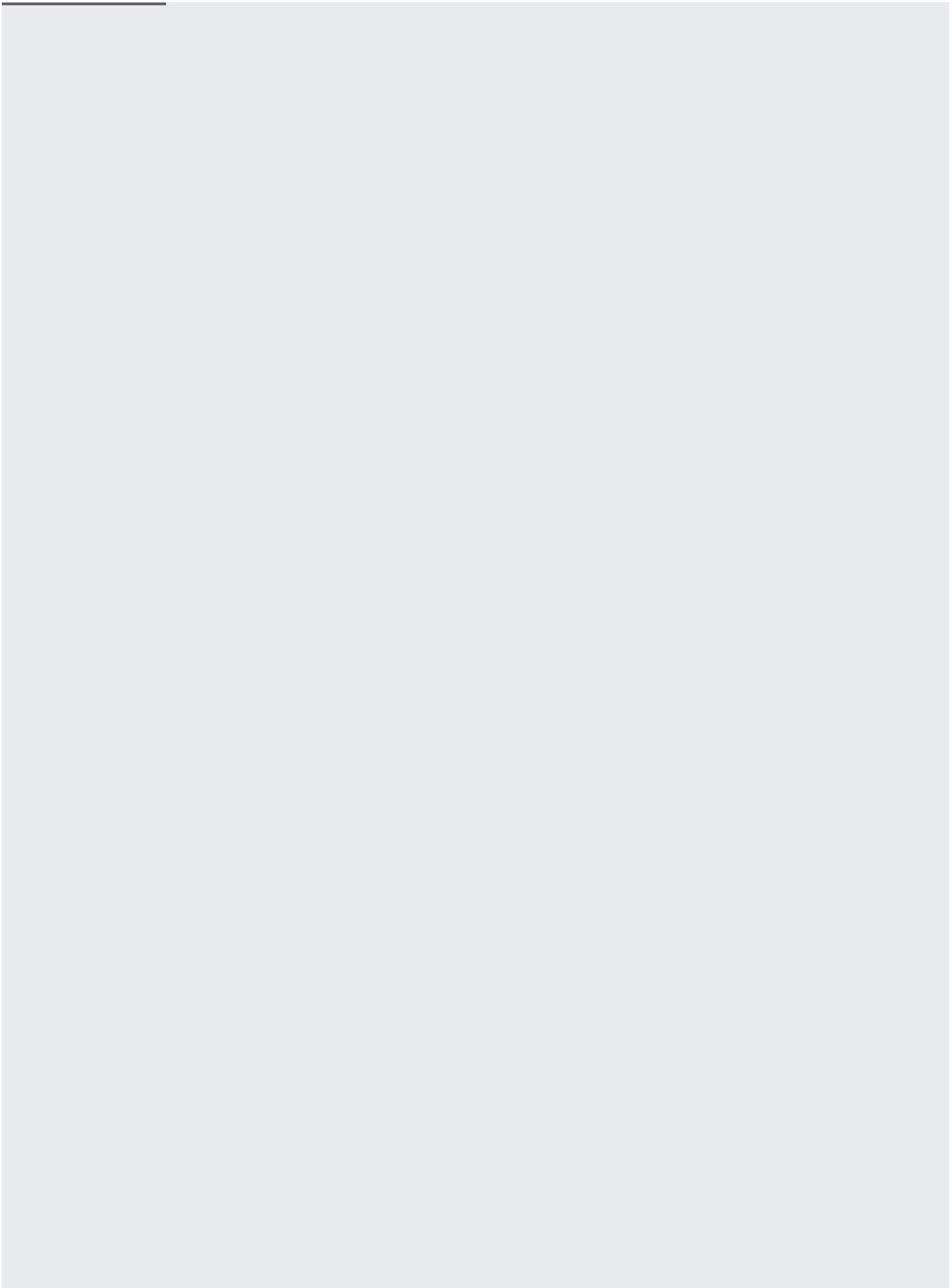
Cloud Spanner provides an interface for executing a body of work in the context of a read-only transaction, with retries for transaction aborts.

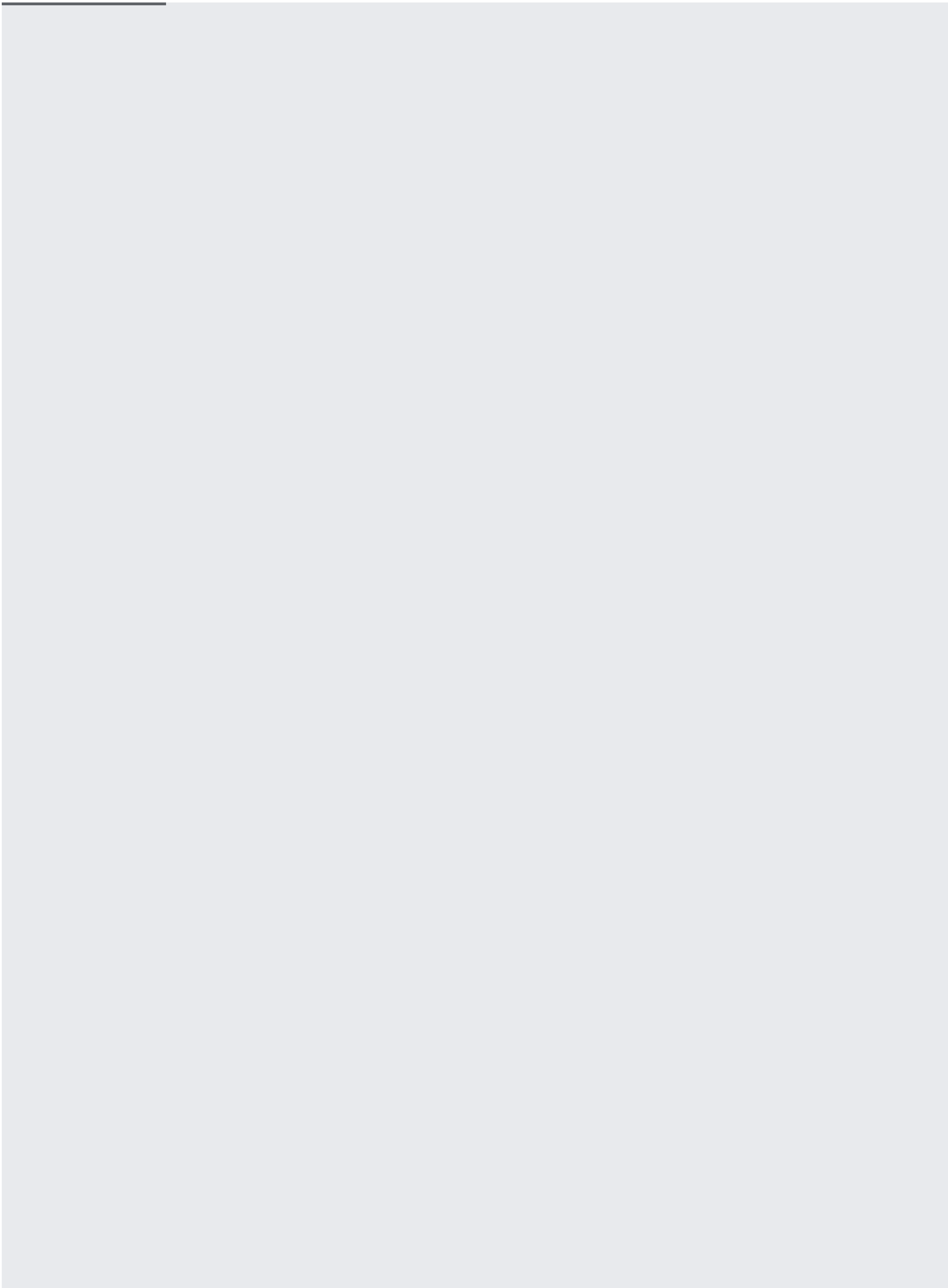
The following shows how to use a read-only transaction to get consistent data for two reads at the same timestamp:

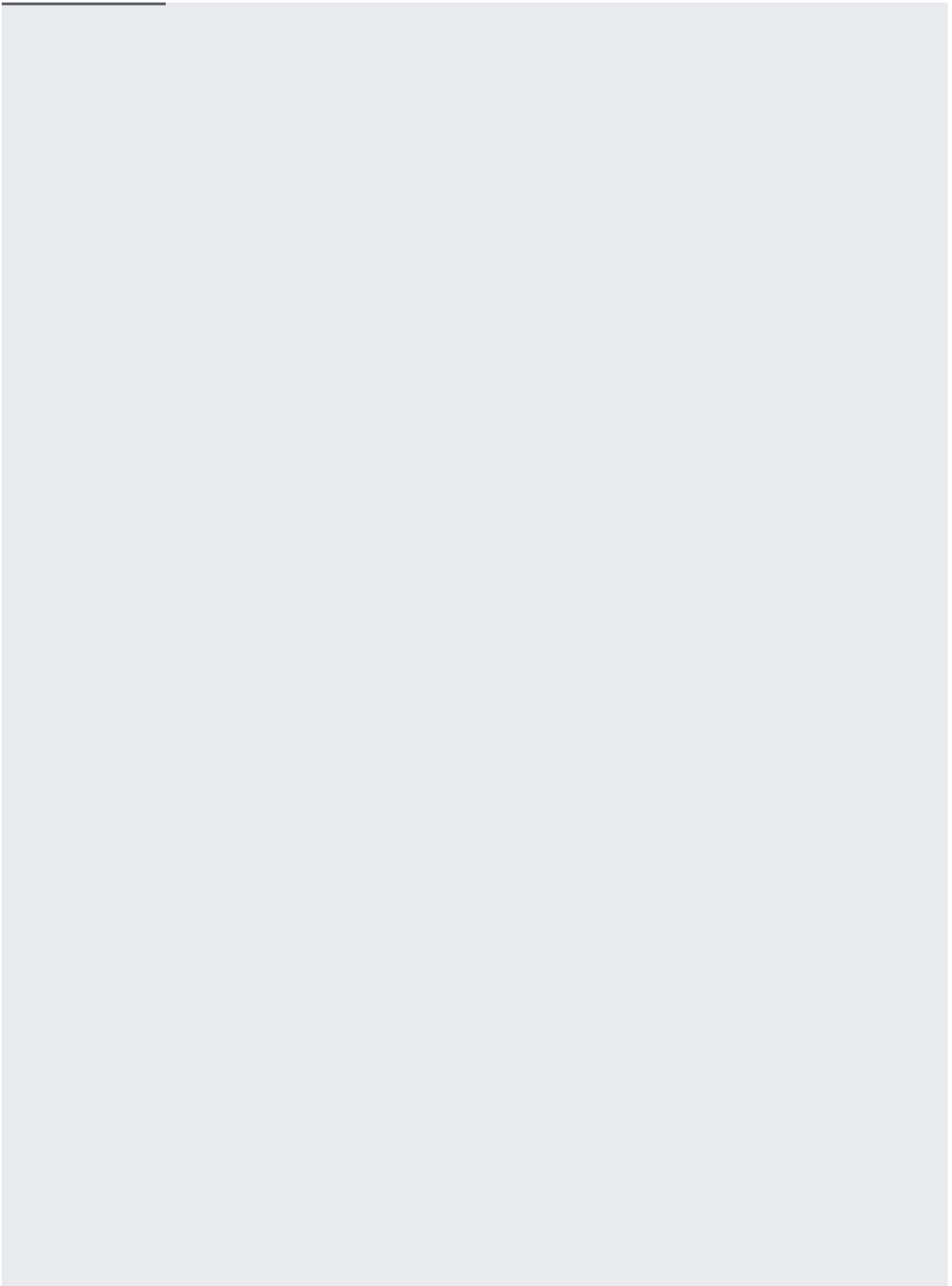


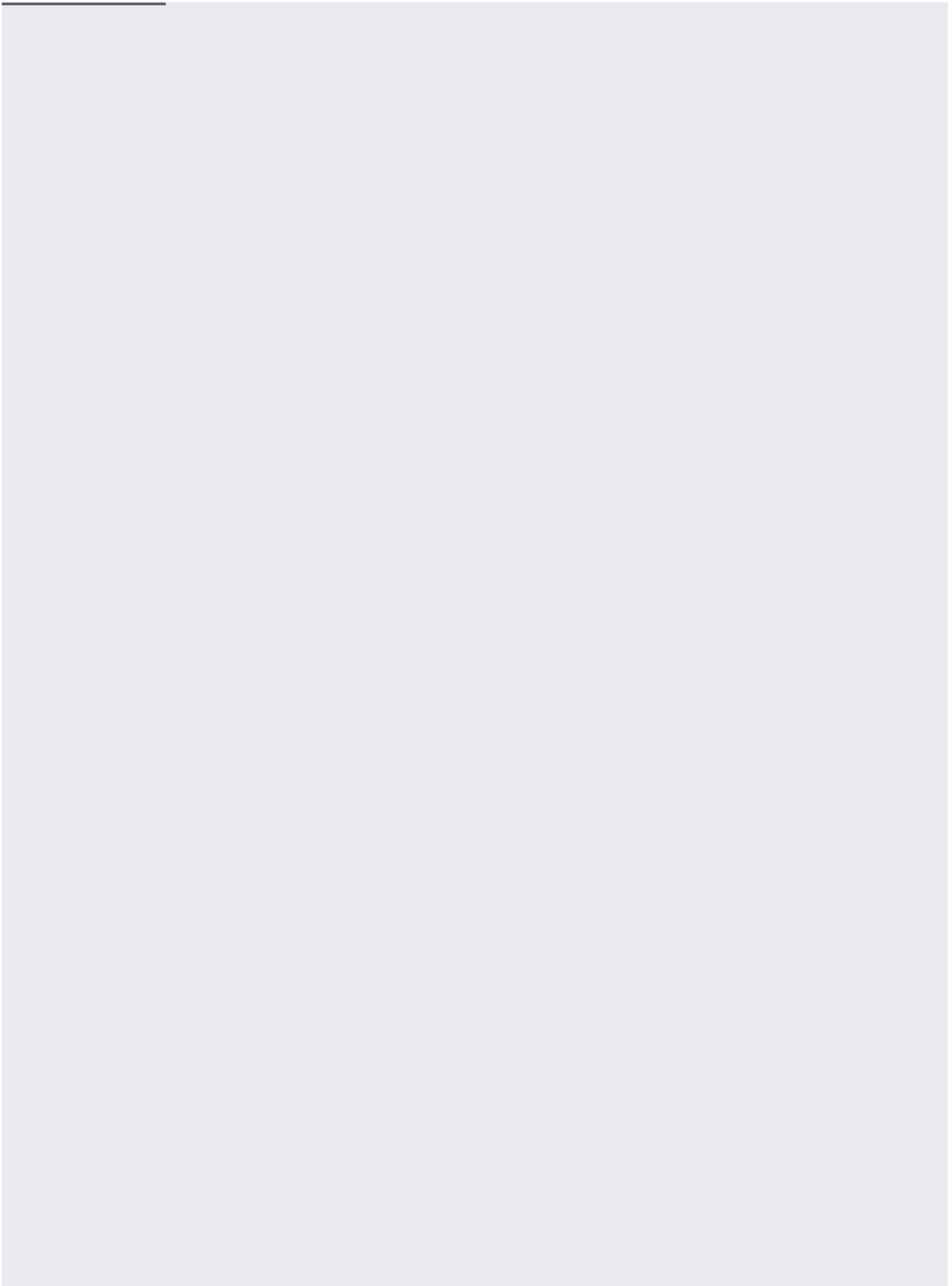












Using Partitioned Data Manipulation Language (/spanner/docs/dml-partitioned) (Partitioned DML), you can execute large-scale `UPDATE` and `DELETE` statements without running into transaction limits or locking an entire table. Cloud Spanner partitions the key space and executes the DML statements on each partition in a separate read-write transaction.

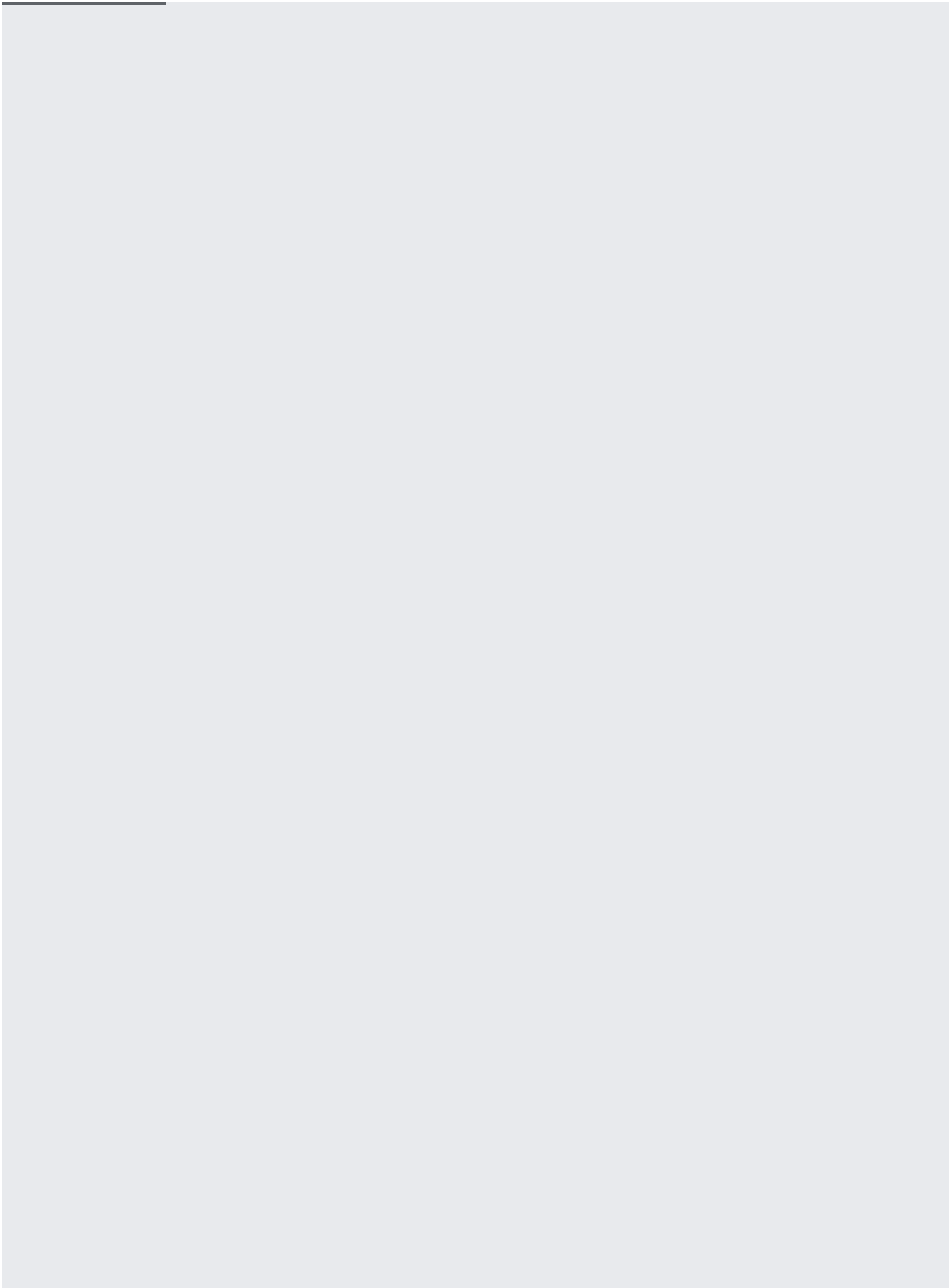
You run DML statements in read-write transactions that you explicitly create in your code. For more information, see Using DML (/spanner/docs/dml-tasks#using-dml).

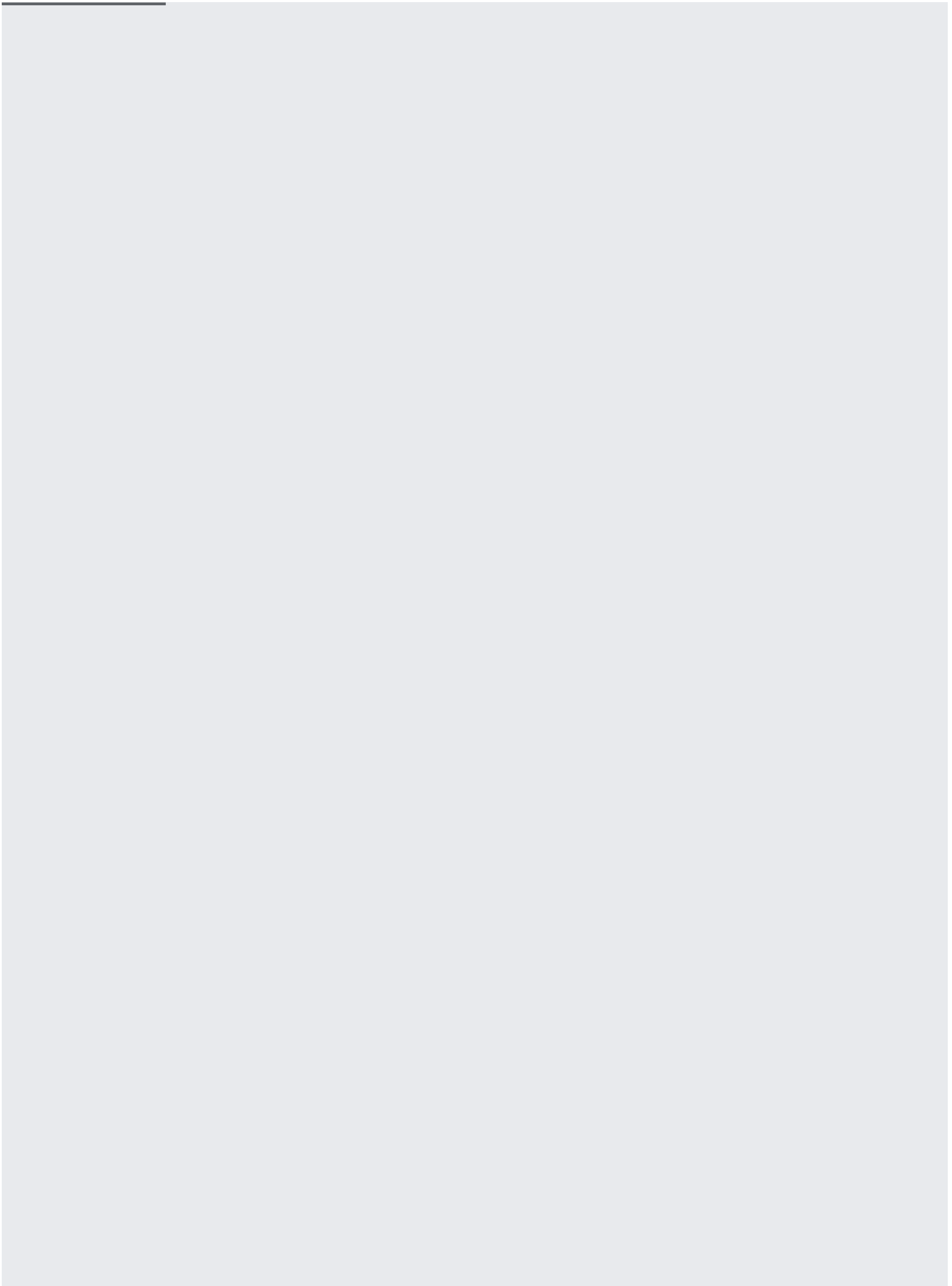
You can execute only one Partitioned DML statement at a time, whether you are using a client library method or the `gcloud` command-line tool.

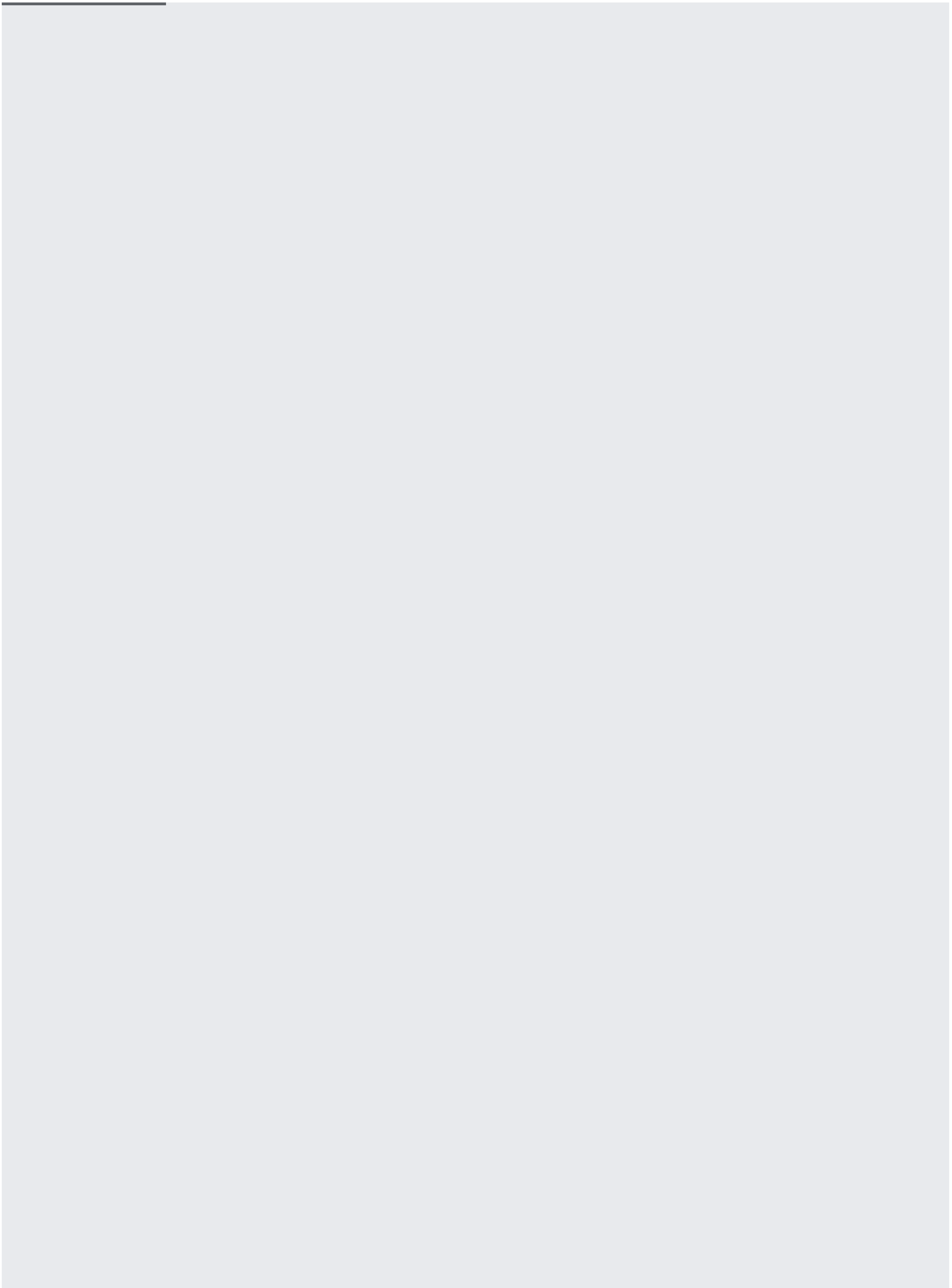
Partitioned transactions do not support commit or rollback. Cloud Spanner executes and applies the DML statement immediately. If you cancel the operation, or the operation fails, then Cloud Spanner cancels all the executing partitions and doesn't start any of the remaining partitions. Cloud Spanner does not rollback any partitions that have already executed.

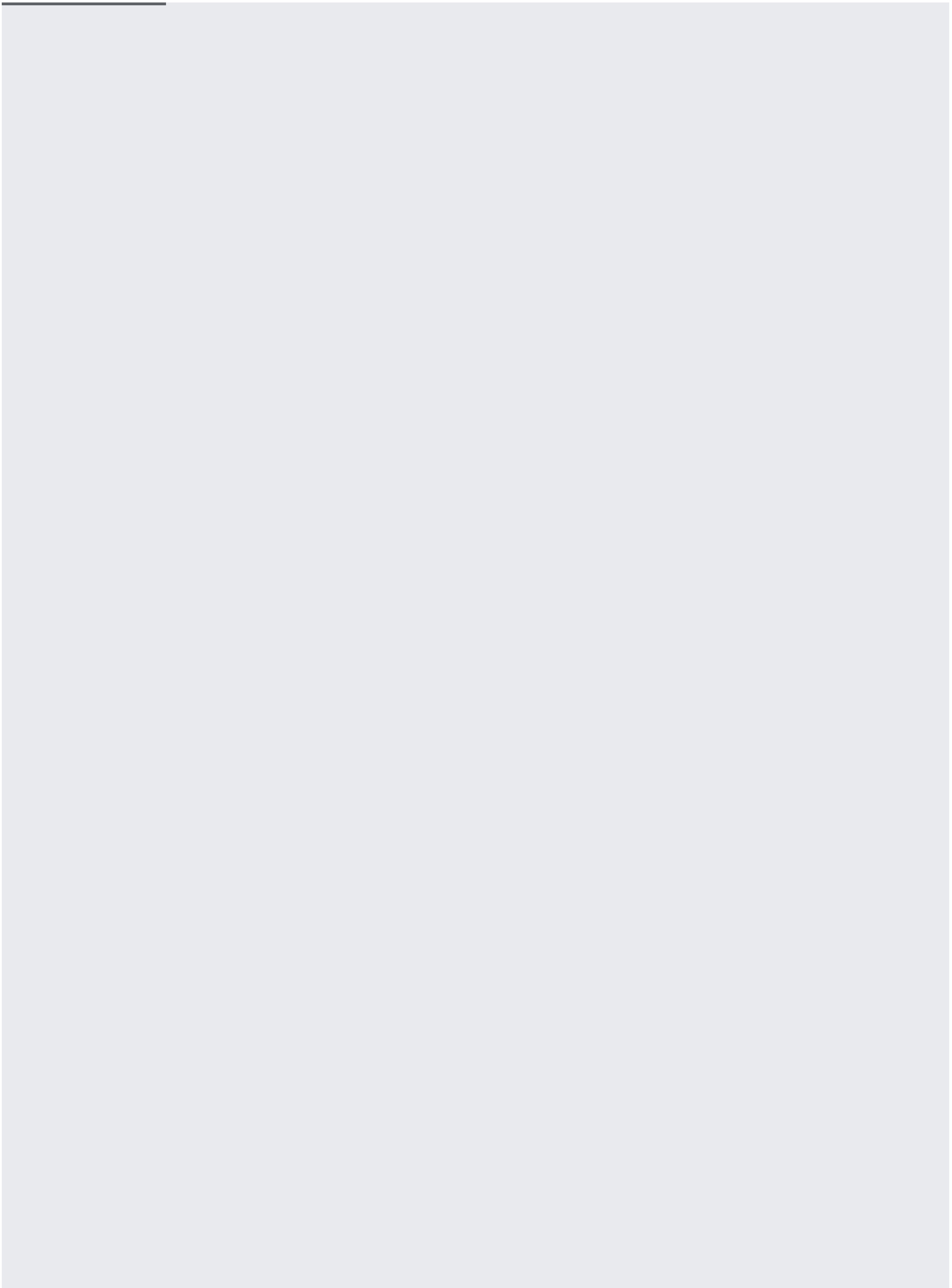
Cloud Spanner provides an interface for executing a single Partitioned DML statement.

The following code example updates the `MarketingBudget` column of the `Albums` table.









The following code example deletes rows from the `Singers` table, based on the `SingerId` column.