Database migration: Concepts and principles (Part 2)

This document discusses setting up and executing the database migration process, including failure scenarios. This document is part 2 of two parts. Part 1 (/solutions/database-migration-concepts-principles-part-1) introduces concepts, principles, and terminology of near-zero downtime database migration for cloud architects who need to migrate databases to Google Cloud from on-premises or other cloud environments.

Database migration setup

This section describes the several phases of a database migration. First, you set up the migration. Then, after you complete the migration and switch over clients to the target databases, you either remove the source databases or, if necessary, implement a fallback plan because of problems with the migration after the switchover. A fallback helps ensure business continuity.

During the migration, you need to give special attention to any schema or data changes that might be introduced. For more information about the impact these changes can have, see Dynamic changes during migration (#dynamic-changes-migration) later in this document.

Target schema specification

For each target database system, you need to define and create its schema. For homogeneous database migrations, you can create this specification more quickly by exporting the source database schema into the target database, thereby creating the target database schema.

How you name your schema is important. One option is to match the source and target schema names. However, although this simplifies switching over clients, this approach could confuse users if tools connect to the source and target database schemas simultaneously—for example, to compare data. If you abstract the schema name by using a configuration file, then giving the target database schemas different names from the source makes it easier to differentiate the schemas.

With heterogeneous database migrations, you need to create each target database schema. This engineering process can take several iterations. Before you can implement the migration, you might need to change the schemas further in order to accommodate your migration process and any data modifications.

Because you will likely create target databases multiple times when you test and execute your migration, the process of creating the schema needs to be repeatable (ideally performed through installation scripts). You can use a code management system to control the version of scripts, ensure consistency, and access the change history of the scripts.

Query migration and execution semantics

Eventually, you need to switch over clients from accessing source database systems to accessing target database systems. In homogeneous database integrations, the queries can remain unchanged if the schemas are not modified. While the clients have to be tested on the target database systems, the clients do not have to be modified because of queries.

For heterogeneous database migrations in general, you must modify the queries because the schemas between the source and target databases differ. The difference might be a data type mismatch between source and target databases. In addition, not all capabilities of the query language available in the source database systems might be available in the target database systems, or the converse. In extreme cases, you might need to convert a query from a source database system into several queries on the target system. In a reverse scenario, where you have more query language capabilities available in the target database than in the source, you might need to combine several queries from the source database into a single query on the corresponding target database.

The semantics of queries can also differ. For example, some database systems materialize an update within a transaction immediately within that transaction, so when the same data item is read, the updated value is retrieved. Other systems do not materialize an update immediately and wait until the transaction commits. If the logic on the source database system relies on the write being materialized, the same logic on the target database can cause incorrect data or even failures.

If you must migrate queries, you need to test all functionality to ensure that the behavior of clients is the same before and after the migration. You can also test at the data level, but such testing does not replace testing on the client level. Clients execute queries from a business logic standpoint and can be tested only on a business logic level.

Migration processes

For heterogeneous database migrations, migration processes specify how data extracted from source database systems is modified and inserted into target databases. Data modifications, such as those discussed in Data changes (#data-changes) in this document, are defined and executed while data items are extracted from the source databases and transferred to the target databases.

With homogeneous database migrations, when the schemas of the source and target databases are equivalent, data modification is not required. Data is inserted into target databases as it was extracted from source databases.

Depending on your database migration system, several configurations might be required. For example, you must specify whether data being modified and transferred must be intermittently stored in the database migration system. Storing the data might slow down the overall migration process but significantly speed up recovery if a failure occurs. You might be required to specify the type of verification. For example, some database migration systems query source and target systems to establish equivalence of the dataset migrated up to the point of the query. Error handling requires that you specify failure recovery behavior. Again, this requirement depends on the database migration system in use.

Needless to say, you need to test your data migration thoroughly and repeatedly. Ideally, your migration is tested to ensure that every known data item is migrated, no data modification errors occur, performance and throughput is sufficient, and time-to-migration can be accomplished.

Fallback processes

During a database migration, the source databases remain operational (unless your migration involves planned downtime). If your database migration fails at any point, you can (in a worst-case scenario) abort the migration and reset the target database to its initial state. After you resolve the failures, you can restart your database migration. The failure and resolution do not affect the operational source database systems.

If failures occur after the database migration is completed and clients are switched over to the target databases, the failure and resolution process might impact clients so that they are unable to operate properly. In the best case, the failure is resolved quickly and downtime for clients is short. In the worst case, the failure is not resolved, or the resolution takes a long time and you must switch clients back to the source databases.

To switch clients back to the source databases, you must migrate all data modifications on the target databases back to the source databases. You can set up and execute this process as a separate, complete database migration. However, because clients are inoperational on the target databases at this point, significant downtime will be incurred.

To avoid client downtime in this scenario, you need to start your migration processes immediately after the original database migration completes. Every change applied to the target database systems is immediately applied to the source database systems. Following this approach ensures that both target and source database systems are kept in sync at all times.

Preparing a fallback from target databases to source databases requires a significant effort. It's critical to decide whether to implement and test fallback processes or understand the consequences of not doing so—namely, significant downtime.

Database migration execution

Executing a database migration involves five distinct phases, which this section discusses. A sixth phase is a fallback, but a fallback is an extreme case and considered an exception to a normal database migration execution.

The process discussed in this section is a near-zero downtime heterogeneous database migration. If you're able to incur significant downtime, you can combine the first three phases (initial load, continuing migration, and draining) into one phase by using either the backup/restore or export/import approach.

A homogeneous database migration presents a special case. With this type of migration, you can use database management system replication functionality (for those systems that support it) that migrates the data while the source database systems remain operational.

The phases discussed here outline an approach that you might need to modify according to the requirements of your database migration process.

Phase 1: Initial load

The starting point is to migrate all data specified to be migrated from all source databases. At the start of the data migration, the source databases have a specific state, and that state changes during the migration.

A tip for starting a migration while changes occur simultaneously is to note the database system time right before the first data item is extracted. With this timestamp, you can derive all database changes from the transaction log starting at that time. In addition, the initial load must read a consistent database state across all data. You might need a short duration lock on the database in order to prevent reading an inconsistent dataset.

This phase consists of the following:

- Noting the database system time right before the database migration starts.
- Executing an initial load migration process that queries the dataset (whether complete or partial) from the source databases that need to be migrated and migrating the dataset. In a relational database model, the initial load migration processes execute queries such as `SELECT`

* or queries with selection, or projection, or both. The migration process performs data modification as specified in the process.

While the initial load migration processes execute, clients typically make changes to the source databases. Because you record the database system time at the start, you can derive those changes from the transaction log later.

The result of the initial load phase is the complete migration of the initial dataset from the source database systems to the target database systems. However, the source and target databases are not yet synchronized because clients likely modified the source databases during the migration. Phase 2 involves capturing and migrating those changes.

Phase 2: Continuing migration

Continuing migration has two purposes. First, it reads the changes that occurred in the source databases after the initial load started. Second, it captures and transfers those changes to the target databases.

This phase consists of the following:

- Starting the continued migration processes from the database system time recorded in Phase 1. The migration reads the transaction log from that time and applies every change to the target database system.
- Executing any data modification. The migration process performs this step as you specify.

Changes that are logged after the database system time are sometimes transferred during the initial load. Therefore, it's possible that those changes can be applied a second time during continuing migration. You need to define your migration processes to ensure that changes are not applied twice—for example, by using identifiers. Suppose a changed data item is transferred during the initial load, and that insert is logged in the transaction log. By applying an identifier to the data item, the migration system can determine from the transaction log that another insert is not required because the data item already exists.

The result of the continuing migration phase is that the target databases are either fully synchronized or almost fully synchronized with the source databases. When a change in a source database system is not migrated, you have an *almost synchronized* database.

Depending on how you configure your database migration system, the discrepancies can be small or large. For example, to increase efficiency, not every change should be migrated immediately, otherwise you can create a heavy load on the source if changes to the source spike. In general, changes are collected and migrated in batches as bulk operations. With smaller batches, fewer

discrepancies occur between the source and target, but your source can incur a higher load if changes are frequently made.

If the batch size is configured dynamically, it is best to synchronize larger batches initially in the continuing migration phase, and then synchronize batches of a gradually reduced size when the databases are almost caught up. This approach makes the process of catching up more efficient and reduces the discrepancy between source and target databases later.

Phase 3: Draining

To prepare to switch clients from the source to the target databases, you must ensure that the source databases and the target database are fully synchronized. *Draining* is the process of migrating remaining changes from the source databases to the target databases.

This phase consists of the following:

- Quiescing the source database systems. This means that no data modifications can occur at the source database, and that the transaction log does not receive additional modification entries.
- Waiting for all changes to migrate to the target databases. This process is the actual draining of changes.
- After draining completes, backing up the target databases in order to have a defined starting point for future incremental backups.

The result of the draining phase is that the source database systems and the target database systems are synchronized, and no data modifications will occur.

To ensure that draining is completed, a "last insert" data item can be written into a source database. Once that "last insert" data item appears in the corresponding target database, the draining phase is complete.

Phase 4: Switchover

After the draining phase is completed, you can switch over the clients from the source to the target databases. We recommend the following best practices:

- Before you enable access to the production database, test the basic functionality in order to ensure that the clients are operational and behave as intended. The number of test cases will determine the actual downtime for your production database.

- Back up the target databases before you enable client access. This best practice ensures that the initial state of the target databases is recoverable.

At the end of the switchover, the clients are fully operational and begin to access production databases (what this document referred to as *target databases* up to this point).

Phase 5: Source database deletion

After you complete the switchover to production databases, you can delete the source databases. It is a good practice to take a final backup of each source database so that you have a defined end state that is accessible. Data regulations might also require final backups for compliance reasons.

Phase 6: Fallback

Implementing a fallback, especially for highly critical database clients, can be a good safeguard against issues and problems with your migration. A fallback is like a migration but in reverse. That is, with a fallback you set up a migration from the target databases back to the source databases.

After you drain the source databases and back up all databases, you can enable migration processes that identify changes in the target databases and migrate them to the source databases before the switchover.

Building these migration processes ensures that after clients make changes to the target databases, the source databases are synchronized and their data state is kept up to date. A fallback might be required days or weeks after the switchover. For example, clients might access functionality for the first time and be blocked because of broken functionality that cannot be quickly fixed. In this case, clients can be switched back to accessing the source databases. Before the clients are switched back, all changes to the target databases must be drained into the source databases.

In this approach, some circumstances require special attention:

- You must design target schemas so that a reverse migration (from target databases to source databases) is possible. For example, if your initial migration process (from source to target) has joins or aggregations, a reverse migration is non-trivial or even impossible. In such a case, the individual data must be available in the target databases as well.
- An issue could arise in which the source databases have transaction logs but the target databases do not provide such a non-functional feature. In this case, a reverse migration (from target to source) has to rely on differential queries. That setup must be designed and prepared in the target database schemas.

- Clients that originally operated on the source databases need to be kept available and operational so that they can be turned on in a fallback. Any functional changes made to clients accessing the target databases must also be made to the clients accessing the source database to ensure functional equivalence.

While a fallback is a last resort, implementing a fallback is essential and must be treated as a full database migration, which includes testing.

Dynamic changes during migration

In general, databases are not static systems (in the sense that the schema never changes or the possible set of valid data values is fixed). However, depending on the schema design and possible data values, changes can occur, and possibly dynamically. The following sections discuss some of the possible changes and their implications for a database migration.

Schema changes

Databases can be categorized into systems that require a predefined schema or that are schema-free or schemaless. In general, systems that require a predefined schema support schema-changing operations—for example, adding attributes or columns in a relational system.

In these systems, you control changes through a change management process. A change management process allows for changes in a controlled way. Any operations that depend on the schema, like queries or data migration processes, are changed simultaneously to ensure an overall consistent change.

Database systems that do not require a predefined schema can be changed at any time. A schema change can not only be done by an authorized user, in some cases it's programmatically possible. In these cases, a schema change can happen at any time. Operations that depend on the schema might fail—for example, queries or data migration processes. To prevent uncontrolled schema changes in these database systems, you must implement a change management process as a convention and an accepted rule rather than by system enforcement.

Data changes

In general, schemas control the possible data values for the various data attributes. Schema-less systems have no constraints on data values.

In either case, data values can appear that were not previously stored. For example, enumeration types are often implemented as a set of strings in database systems. On a programming language

level, these might be implemented in clients as true enumeration types, but not necessarily so. It is possible that a client stores what it considers a valid enumeration value that other clients do not consider as valid. Furthermore, a data migration process might key its functionality off enumeration values. If new values appear, the data migration process might fail.

Another example is found in the storage of JSON structures. In many cases JSON structures are stored in a datatype string; however, those are interpreted as JSON values upon access. If the JSON structure changes, the database system does not detect that; data migration processes that interpret a string as a JSON value might fail.

Migration process changes

Change management during an ongoing database migration is difficult and complex and can lead to data migration failures or data inconsistencies. It is optimal that required changes are delayed until the end of the draining phase, at which point the source and target database systems are synchronized. Changes at this point are then confined to the target databases and their clients (unless a fallback is implemented as well).

If you need to change your migration process during a data migration, we recommend that you keep the changes to a minimum and possibly make several small changes instead of a more complex one. Furthermore, you might consider first testing those changes by using test instances of your source and target databases. Ideally, you load the test source with production data that you then migrate to the test target. Using this approach, you can verify your proposed changes without affecting your ongoing production migration. After you test and verify your changes, you can apply the changes to your production system.

For changes to be possible during an ongoing data migration, you must be able to stop the data migration system and restart it, possibly with modified data migration processes. In that case, you don't have to start from the very beginning with an initial data load phase. If the data migration system supports a test migration run feature, you can use that as well.

We recommend that you avoid changing schema, data values, or data migration processes during a data migration. If you must make changes, you might consider restarting the data migration from the beginning to ensure that you have a defined starting state. In any case, it's paramount that you test using production data, and that you back up your databases before you apply changes so that, if needed, you can reset the overall system to a consistent state.

Migration failure mitigation

Unexpected issues can occur during a database migration. The following highlights a few areas that can require preplanning:

- **Insufficient throughput.** A migration system can lack sufficient throughput despite load testing. This problem might have many causes, such as an unforeseen rate increase of source database changes or network throttling. You can prepare for this case by preparing additional resources for dynamic scaling up or scaling out of all involved components.
- **Database instability.** Source databases or target databases can exhibit instability, which can slow down data migration processes or intermittently prevent access. Data migration processes might need to recover frequently. In this case, an intentional HA or DR switchover might address the issue. A switchover changes the non-functional environment (machines and storage) and might help mitigate the problem. In this case, you need to test the switchover and the database migration recovery processes to ensure that the switchover does not cause data inconsistencies in the target databases.
- **Transaction log file size exhaustion.** In some cases, transaction logs are stored in files that have an upper limit. It is possible that this upper limit is reached and the database migration then fails. It is important to understand which parts of a database system can be dynamically reconfigured to address resource limitations as they arise. If certain aspects cannot be dynamically configured, initial sizing must be carefully determined.

The more you make upfront testing realistic and complete, the more likely it is that you'll find potential issues to address in advance.

What's next

- Check out the following resources on database migration:
  - Migrating from PostgreSQL to Cloud Spanner (/spanner/docs/migrating-postgres-spanner)
  - Migrating from an Oracle® OLTP system to Spanner (/solutions/migrating-oracle-to-cloud-spanner)
  - Migrating a MySQL cluster to Compute Engine using HAProxy (/solutions/migrating-mysql-cluster-compute-engine-haproxy)
- See Database migration (/solutions/database-migration) for more database migration guides.
- Try out other Google Cloud features for yourself. Have a look at our tutorials (/docs/tutorials).

Last updated 2020-05-29.