Online Database Migration by Dual-Write: This is not for Everyone

(to be more precise: for almost no-one)

Christoph Bussler Follow
Jun 23 · 42 min read

tl;dr

Online database migration is an important — if not the most important — approach when migrating between schemas, between databases, or between data centers or clouds. There are many variations to online database migration (aka, zero downtime database migration): a popular one is dual-write and in my opinion a — if not the most — "dangerous" one when data consistency is paramount. This blog rationalizes two different online database migration variations (dual-write and change data capture (CDC) based) to highlight the risks involved when using dual-write online database migration; and the length of this blog reflects the "danger" when embarking on it.

My goal is to have you think twice, three or even four times before taking the dual-write fork in the database migration road and point out only the toughest technical problems you will have to face and to solve.

Online database migration

In a nutshell, the goal of online database migration is to migrate from one or more source databases to one or more target databases with minimal source database downtime and — as a consequence — minimal client downtime. Ideally the downtime is zero, even though it is not possible in all cases, as clients have to reconnect to the target database after the online migration is complete. Typical migrations are from an on-premises data center to a cloud or between clouds, or even within a cloud.

Key requirements and expectations

In order to make migration predictable and reliable with as little effort as possible, the following key requirements are expressed in many database migration projects:

- **Migration is complete**: all data in the source database is completely migrated (or a defined subset of the source database if not all data is to be migrated)
- **Migration is consistent**: the target database is fully consistent (transactional consistency), no data loss, no data duplicates, and no transaction reordering
- **Migration is repeatable**: target databases can be recreated and migration started from the beginning (for example, to mitigate unforeseen errors)
- **Migration does not impact client operations**: clients operate without any impact during ongoing migration (for example, no downtime)
- **Migration does not impact client code**: client code does not have to be changed to support migration to avoid any implementation effort (aka, design, testing, etc.)
- **Migration can be tested**: testing of migration must be possible, especially of migrations where the source schema is different from the target schema, without impacting the production clients accessing the source databases

Brief terminology

- **Source, target database**. The source database has the current data set and the target database is the database the data from the source database is migrated to. The target database might or might not have the same schema as the source database. If the schemas are different in the source and target database, data transformation is required.
- **Primary database**. At any point in time, either the source database or the target database is the source of truth, aka, the primary database accessed by clients. In a given database migration there will be the point in time where the target database will be made the primary and becomes the authoritative source of truth for clients.

Before discussing the dual-write variant to online database migration the CDC (change data capture) online database migration process (that does not rely on dual-write) is summarized next as a baseline for comparison. The CDC online database migration process is widely used and a well-tested variant. Briefly outlining the CDC online

database migration process makes the comparison between the two variants easier later on.

Baseline: CDC online database migration process

The basic architecture of CDC online database migration is outlined in this diagram:

CDC online database migration architecture

One or more clients operating on the source database are not affected by the separate migration system migrating the data to the target database. The migration system relies on transaction logs to observe database changes. Once migration completed, the clients can be switched over to the target database.

The CDC online database migration process (in very detail discussed here: [5] [6]), has the following steps (assuming one source and one target database for the discussion in this blog, multiple source and target databases are supported equally in a production environment):

- **Create a target database**. This might be the same or a different database engine compared to the source database, and it might have the same or a different (modified schema) compared to the source database.
- **Create the migration logic using migration technology**. This can be built-in functionality a database provides (like logical replication), or it can be a separate migration system like Striim or Fivetran (to name two examples from a large variety of available migration technologies).
- **Perform an initial load**. This is the transfer (including possibly data transformation) of a transactionally consistent source database snapshot to the

target database.

- **Perform continuous migration using CDC**. Change data capture (CDC) is a mechanism that reads the database's transaction log and applies committed source database transactions in commit order exactly once to the target database (including possibly data transformation). This process starts from the database time the initial snapshot was taken to ensure all concurrently ongoing transactions during the initial load are captured and migrated. At the end of the blog an alternative to transaction log access is discussed in case the source database does not expose those.

- **Complete draining**. When the day and time of the cut-over of clients from the source database (former primary) to the target database (current primary) nears, write access to the source database is stopped. The remaining transactions from the transaction log that have not been migrated yet to the target database have to be migrated ("drained") to the target database. If configured carefully, the draining takes place in a matter of seconds.

- **Cut-over clients**. At this point the target database has all data from the source database, and their state is equivalent wrt. transactional consistency, clients connect to the target database, access it and continue the production workload.

The key is that during the initial load, and the continuous migration, clients can continue to access the database without restriction and without modification of any client code. This means that the clients and the database migration are ongoing concurrently. The only unavailability of the source or target database is during the draining and cut-over phase, which can be extremely short. Depending on the strategy, additional testing can take place on the target database before clients reconnect to the target database adding to the client downtime for reasons of safety.

The important aspect to note is that clients are not impacted by the migration and their code base does not have to be modified at all in order to accomplish the near-zero downtime database migration.

Also important to note is that this variant supports migration between different database engines and schemas without having to modify the client accessing the source database. Of course, the client for the target database has to be aware (code-wise) of the difference of the schema and the difference in query and transaction processing on the

target database. The new version of the client for the target database is implemented and can be tested during the online database migration without impacting the source database.

The above process does not describe an optional fallback process (for brevity in context of this blog) that is important for many migrations in order to deal with unforeseen problems or issues even after the migration was completed a few weeks after the cut-over to the new target database.

Dual-write online database migration process

The overall architecture of the dual-write variant is shown in this diagram:

Dual-write online database migration architecture

One or several clients read and write from a source as well as target database and implement the database migration functionality.

A variety of references outline a dual-write migration variant to online database migration, for example, Online migrations at scale [1], Safe Database Migration Pattern Without Downtime [2], or Zero Downtime Table Migrations using a Double Write Methodology [3]. Please note, [2] is intermittently available at the time of this writing and you might have to go to the Internet archive to retrieve the content.

The overall dual-write migration process is as follows based on the above reference [2] (as this reference has the most detailed set of steps — all steps are directly cited from

[2]). Other resources are following the same or a similar process:

- Step 1: Build and deploy the “new” database schema onto production.
- Step 2: Add a new DAO to your app that writes to the “new” database.
- Step 3: Start writing to the “new” database but use the “old” one as primary.
- Step 4: Enable the read path. Change the feature toggle to enable reading from both databases.
- Step 5: Making the “new” database the primary one. Change the feature toggle to first write to the new database (you still read from both but now prefer the new DB).
- Step 6: Stop writing to the “old” database (read from both).
- Step 7: Eagerly migrate data from the “old” database to the “new” one.
- Step 8: Delete the “old” DAO.

The steps are explained and discussed in detail in the references and would take up too much space to cite here verbatim, however, before continuing reading this blog below, I encourage you to take a few minutes and read at least [2] in case you want to understand the outlined steps in more detail.

[3] outlines the special case of migrating between tables within the same database, not migrating data between databases. While some aspects are similar, some are different. The differences are called out separately below.

Preliminaries — system properties

Unless you are migrating tables within the same database from source to target tables as discussed in [3] (which I would consider a rare case based on my experience) and you are instead migrating between databases in different database servers (instances), the following system properties are very important to be aware of:

- **Distributed transactions**. Cross-database transactions are only available if you use a distributed transaction manager that can coordinate between the two servers chosen and the servers can participate in the distributed transaction protocol. In all

other cases, transactions do not span the two databases and any access of both databases is not serialized (and therefore not automatically ensuring data consistency). Neither sequential nor concurrent access of both databases is serialized by the two databases involved).

- **Consistent backup**. Since the two databases are independent of each other, it is not possible to have a consistent backup of both databases unless you stop write access on both when creating backups. Two backups, one for each database, are only consistent when there is no change during the backup in any of the databases.

- **High availability and disaster recovery**. The databases are independent of each other wrt. high-availability or disaster recovery. If one database (source or target database) fails over then the fail over process is independent of the other database. There is in general a possibility of downtime as well as a set of lost transactions during high-availability or disaster recovery that might violate data consistency (in the absence of distributed transactions). In this case both databases are out of sync and have to be brought in sync first before continuing migration. For example, the fail over of the target database loses transactions that are present in the source database. The lost transactions have to be determined and migrated again.

- **Networking and latency**. The two databases in general might run in different environments with different network properties (like throughput or routing structure). This difference might impact latency between clients and the database access interface. In the absence of distributed transactions this might lead to unintentional reordering of transactions between the source and target database from a client perspective leading to data inconsistency.

In high-volume production environments above properties are important as they might affect how to implement online database migration so that any issue or problem at runtime does not compromise data integrity and data consistency. In my opinion, addressing above issues are complex and a hard engineering problem in context of the dual-write variant.

Preliminaries — sample use case: user and addresses

The following use case based on a relational model is used to illustrate specific aspects during online database migration. It is kept simple for the purpose of keeping this blog

short and helping illustrate only a basic set of query behavior. In a production database with a more complex data model (relational or non-relational) additional query behavior might be important to analyze and to implement.

Sample use case: user and its addresses

The schema consists of two entities, user and address, and a user can have zero, one or more addresses. An address cannot exist on its own and is part-of a user. The address.user_id has a foreign key relationship with user.id. The columns characterize a user and account respectively, and are kept to a minimum for illustration.

While the use case is expressed in the form of a relational schema, the following discussion applies to all data models as the relational model is capable of effectively representing all other known data models (although not necessarily efficiently).

Discussion of dual-write database migration process

The following discussion is guided by data consistency as the guiding principle. Any choice or argument is made to ensure that a target database after migration and before clients are switched over to it is complete and consistent. In the following, each discussion topic discusses both, the dual-write variant as well as the CDC variant.

Client code modification

Dual-write requires client code modification (see Step 2 above). The client code has several responsibilities: implement the business logic, access the source database, access the target database, and in case the source and target schema are different, it has to implement the transformation between the two databases and transformation of results before returning the results.

Each code modification requires testing and this means that tests and test data sets have to be developed and maintained that confirm that no regressions take place (in the business logic as well as the migration logic (!)). The difficult aspect is that in case of dual-write migration the target database is continuously changed and is not reset for every migration or business logic code change, so any test database has to be maintained accordingly.

Code modifications can be required based on business logic improvement or migration logic changes. The references indicate that dual-write migration can take weeks or months. In this time frame business logic modifications might be required in addition to the necessary migration logic changes. Since these are orthogonal functionality, their independence and isolation has to be tested as well.

If source and target databases have a different schema, as soon as the target database is made the primary source of the data, the client code has to work on the target schema and therefore has to be modified accordingly. At this time (after the target database is made the primary) the client code accesses the target database first, then the source database. Any transformation has to now be from the target to the source schema. This is the reverse direction from before when the client was writing to the source database first.

The target database might have parts of its schema in place for transformation purposes only and those parts will have to be removed [1]. It is possible to delay the target schema changes in the client code to after the migration is completed to limit code changes during the migration itself.

CDC database migration: client code accessing the source database does not have to be modified, and client code accessing the target database after migration has to be modified only in case the target database has a different schema from the source database. That modification of the client code can be implemented and tested

independently and separate from the database migration (and the version of the client code accessing the source database).

Multiple concurrent clients

If the source database is accessed by several clients concurrently, the dual-write migration has to be implemented and coordinated across all clients in order to ensure that the clients do not conflict with each other while accessing the correct and consistent data. This requires migration logic to deterministically find out if data a client accesses is to be migrated or was already migrated.

CDC database migration: client logic does not have to be modified; any number of concurrent clients can access the source database.

Data access path management

A client has to be able to determine if data it is about to access was already migrated or not (Step 3). If not, then it has to access both, the source and the target database. There are three operations that have to be modified for database migration: insert, update, delete.

- Insert: data has to be inserted into both databases
- Update: data is to be updated in the target database (in addition to the source database) only if the data was migrated before
- Delete: data is to be deleted in the target database (in addition to the source database) only if the data was migrated before

The references do not outline how the data path management is implemented that allows a client to determine if migration for data took place already or not. It could be a dynamic query to the target database to find out if the data in question is present. Another alternative is to have a lookup function on managed metadata that for a given identifier (e.g. primary key) returns if the data was migrated or not. If the lookup function is based on a persistent mapping (e.g. in a database table) then concurrent queries to the same primary key might fail if one of the queries migrates the data and a subsequent query tries to update it as the code executing the second query might not have picked up the change in the access path.

No matter the path management implementation, the lookup must be deterministic for the client to get the correct answer. Below more complex querying cases are discussed in more detail based on the example use case.

Since the roles of the databases change (Step 5) the access paths for read operations has to be changed as well as the write path so that the code accesses the primary database first.

CDC database migration: since clients are not modified and access only the source or the target database, no data access path management is required.

Process

When data consistency is the topmost goal, it must be clear throughout the whole migration process, which of the databases that are part of the migration is the primary database that contains the consistent and complete source-of-truth state at any point in time.

In any migration process use case, the source database is the primary before the migration begins, and the target database is the primary after the migration completed. Which one is the primary during the migration process when the migration is ongoing? In the CDC database migration variant, the source database is the primary throughout the whole migration process. Only after it is completed the target is made the primary short before the clients are cut over from the source to the target. It is clear cut.

In the dual-write variant it is actually not clear what the situation is based on the above cited process. Step 3 clearly states that the source database is the primary database. Step 5 states that the target database is made the primary (and acknowledges that the source might become inconsistent as writes to the source are “best effort” only). However, only at step 7 back fill takes place, aka, not yet migrated data is migrated from the source to the target. This means that at step 5, even though the target is made the primary, it does not have the complete data set. Furthermore, step 6 states to stop writing to the source database. At this point no database has the complete and consistent data set as both databases converge from each other. Only after step 7 (back fill) the target database is complete in the sense that all data is migrated. There are possibly business logic errors being caused by this variant for the time when no database has the complete and consistent data set (see next paragraph).

From my viewpoint I would ensure during any migration process that one of the databases is the primary at any point in time holding the complete and consistent data set in order to actually achieve and provide data consistency.

The references refer to “changing the read/write paths” to change the order of which database is written to first or read from first. When data consistency is important, the order of which database is accessed first does not matter as the primary must be complete and consistent.

Details of migration logic functionality

While the initial expectation is that the dual-write variant only requires the same access logic (read, insert, update, delete) executed twice, once on the source and once on the target database, this does not hold true when starting addressing the various detailed database access use cases. In the following I will discuss some of them in order to illustrate the complexity of required changes to the client code. Every time a dual-write takes place the access path management might have to be updated depending on its implementation (see the discussion above).

The goal is to migrate the data from the source to the target database while preserving data consistency. At any point, the data must be consistent, and all read access must return a correct result — the presence of a target database must not introduce inconsistencies.

First, inserts are discussed:

- **Primary key row insert (without relationship)**. A single row insert by primary key can take place in each, the source and target database, in the same way. In the use case, a new user can be added as a single insert.
- **Primary key row insert (with explicit relationship)**. A single row insert by primary key that has a foreign key relationship can be different for the source database and the target database. In the use case, a new address for a given user is a single insert into the source database. In the target database, however, the code has to first check if the user is present since the address has a foreign key relationship to the user. If not, the user has to be migrated from the source before the address can be inserted.

- **Primary key row insert (with implicit relationship)**. A single row insert by primary key that has an implicit relationship can be different for the source database and the target database. An implicit relationship is not defined by the schema, but managed by the client by means of code. If an insert requires related data to be present (one or more relationships), the code for the target database has to check for the data presence and has to ensure its migration if it is not in the target database. Implicit relationships are difficult as those cannot be found by examining the database schema, but only by understanding the business logic in the client code.

- **Primary key row batch insert**. Batch inserts on the target only work for those insert use cases that do not involve a relationship. As soon as a relationship is involved checks have to be performed (see discussion in bullet points above).

Next, updates are discussed:

- **Primary key row update (all columns)**. If all columns are updated, in the target database a check has to ensure that the row is present first. If not, it has to be migrated first, and then updated. Alternatively the update can be converted into an insert. It depends on additional context (like database triggers or predicates in the update statement or relevance for transaction log content) which way is chosen.

- **Primary key row update (subset of columns)**. If a subset of columns are to be updated, the row has to be migrated first, before the update takes place.

- **Primary key row update with relationships**. A row update with explicit or implicit relationships (like a foreign key) has to ensure that the mandatory related data (one or many) is present first in order to not violate the referential integrity. In addition, the row to be updated has to be present or migrated first.

- **Primary key row batch updates**. Batch updates on the target database only work if all rows that are to be updated are present. Any update requiring additional logic cannot be part of a batch update.

The next discussion is about deletes:

- **Primary key row delete (without relationship)**. A delete can take place on the target database if the row is present or not. If it is not present, the delete will

succeed. However, context matters: if transactions logs on the target have to be consistent with those on the source as well, then a delete has to be preceded with migrating the data first so that the delete actually takes place on the target database. Other examples are triggers that might require the delete to actually take place.

- **Primary key row delete (with implicit or explicit relationship)**. A delete or a row might have to trigger the delete of additional rows. For example, if a user is deleted, all addresses have to be deleted. In case of relationships specified in a schema the database supervises those; in case of implicit relationships the code has to ensure proper traversal and deletion itself.
- **Primary key row batch deletes**. As before, a batch delete only works if no additional logic has to be executed when deleting.

Next up is read access:

- **Read from source as the primary database**. If the source database is the primary database all read operations are executed on the source database as it has the source of truth and represents the consistent database state.
- **Read from target as the primary database**. The target database can only be the primary database if it has the complete set of data that is consistent as well. Until the source is switched off, this implies that both, the source and the target are 100% consistent (aka, a perfect copy of each other or are perfectly in sync wrt. data consistency in case of different schemas). The above process taken from [2] does not accomplish this: it makes the target database the primary (step 5) without having migrated all data (step 7). I think the term “primary” in the reference refers to which database is written to first, and has no meaning wrt. data consistency — which is the customary use of the term “primary” in database architectures.
- **Read from both databases?** Reference [2] mentions obtaining read results by accessing both databases and combining (“merging”) the results based on timestamps in the schema (step 4). This has several implications: first, a timestamp has to be added to every table and the timestamp has to be taken from the same clock across two databases. This is not accurately possible unless the databases are within the same database instance. Second, this means that neither the source nor the target database have a consistent (and complete) data sets. It is not clear to me

how it can come to this situation and how to manage consistency (I suspect the discussion in step 3 means that write failures to the target database can be ignored). However, since the references [2] mentions it, it is important to mention it here because of the data consistency implications.

Operations beyond update, insert, delete and update:

- **Aggregation**. Any form of aggregation depends on a complete data set (or a complete subset that is in scope of the aggregation queries). If a query counts the number of unique addresses, all users and all addresses have to be accessible. In a dual-write situation this is only supported and possible if at any point in time the source or the target is the primary database that is consistent and complete. If the dual-write variant cannot ensure a single consistent database at any point, aggregation queries will return inaccurate results during the data migration. Alternatively, aggregation queries have to be rewritten to derive their result by accessing both databases.
- **Join**. Like in case of aggregation operations, a join only provides correct data on a consistent and complete data set.

Side effect operations:

- **Stored procedures, triggers, etc**. In some cases database schemas have stored procedures, triggers, or additional operations that are database local. The individual implementation has to determine if those can operate concurrently on the source database and the target database. If they cannot work concurrently on both (meaning, on incomplete data on the target database), those will have to be switched off on the target database (non-primary database) until it is data complete and consistent. However, if switched off, any data migrated, updated, etc. will not cause any side effect that is required for consistency (e.g., a stored procedure updating the database itself). This will make the dual-write migration an impossible variant to use.
- **External database access**. Some database schemas use triggers to communicate with downstream systems, or use external tables to access upstream systems. As before, the specific implementation has to determine if those can be enabled on the target database during migration and if consistent behavior is taking place.

- **Transaction logs**. Some use cases rely on transactions logs. Changes made in the database are observed and downstream functionality is triggered. In order for these use cases to work it is necessary that operations in the source as well as the target database are executed in the same way so that the transaction logs are equivalent as well. For example, as discussed above, an update on the target must not be converted into an insert if the data was not migrated before.

Relationships:

- **One-level relationships**: As discussed above, one-level relationships — relationships between two tables — that are implicit or explicit have to be addressed by the code accessing the target database.
- **Multi-level relationships**: Like one-level relationships multi-level relationships have to be taken care of by the client code accessing the target database. In a schema with many multi-level relationships complex traversals might be required to ensure that the required data is migrated during operations that require the data to be available. In the worst case, if the data set represents a tightly connected graph, the first operation requiring migration might have to migrate over all data before it can continue (e.g., insert or update).

Reference [3] points out that it might be impossible to find all operations accessing the source database in the code base even though the target database is the primary. If the source database is taken offline, this might cause errors in production. In my opinion I would consider refactoring the client code first before starting the database migration to ensure that all operations can be enforced to execute exclusively on the target (by means of e.g. a code switch). One type of refactoring would be the explicit access path management for both databases separately. A further refactoring would be to isolate all access operations in a separate architecture layer that can be configured and reviewed as a separate component.

CDC database migration. Except for side effect operations that write access to external data sources none of the above issues are present in the CDC database migration variant. Operations that write access to external data have to be analyzed in order to understand if switching those off during migration avoids any downstream system issues (since the source database is taking care of it already). If switching off is not

possible, refactoring of those might be required in order to make the database migration possible.

Tables without primary keys

It is possible that tables exist that do not have primary keys. This makes database migration difficult or impossible when data consistency is to be guaranteed as tables without primary keys cannot be analyzed (e.g., does a row exist? Can the same values be in two or more rows?) without understanding the code operating on those tables.

The recommendation is to ensure that all tables have primary keys and refactor the code and the schema before embarking on a database migration. A second, less preferable approach, is to migrate tables without primary keys completely first during a client downtime period before the regular migration starts as a preliminary setup. This ensures that client logic does not have to do the migration of key-less data, but only execute subsequent changes (insert, update or delete).

Consistency

Consistency is discussed separately because of its inherent complexity in case of the dual-write variant. The following discussion assumes that one of the two databases is the primary at any point in time, and that once the target database is the primary, the source database is consistent with it (true copy). If this is not the case, then consistency is not given by the database state of one database (its content), but by read access operation results across databases and only if their combined data state is consistent (see discussion above).

- **Partial transaction abort (transaction coordination)**. As discussed in the preliminaries above, transactions cannot span the source and the target database (unless migration between tables within the same database is implemented). This means, for example, that an update of a primary key row in the source and the target database are two different transactions: one on the source and one on the target database ([2] points this out explicitly). In order for the databases to be consistent, both transactions have to succeed or fail. While distributed transactions would guarantee this behavior, in their absence the client code has to ensure this behavior. This means that a client has to be able to recover from a source transaction failure or a target transaction failure if the transaction of the other database succeeds. In

addition, this recovery has to work in presence of concurrent transactions on the same object (primary key) while transaction failures are taking place (this is core database logic of concurrency and serializability). If implemented correctly, it replicates the two-phase distributed transaction protocol in the client code.

- **Sequential transaction reordering**. If a client issues two updates right after each other on the same user (same primary key), four transactions will be executed. However, in the absence of distributed transactions, the order of the two transactions on the target database can be the reverse from those on the source database, leading to data inconsistencies, as the user's state would different in the source and the target database. This can only be avoided if the client code explicitly ensures that sequential transactions on the same primary key are executed in that order on both, the source and the target database.

- **Concurrent transaction reordering**. If two different clients issue for example an update on the same user (same primary key) concurrently, four transactions are executed. Since the transactions are independent of each other, they can be executed on the source and the target database in a different order. Unless client code implements cross-client coordination, data inconsistency can occur as the state of the source and target database can be different.

- **Concurrent transactions requiring data migration**. One of the use cases is that two concurrent updates are initiated by clients on a user (same primary key) whereby the user has not yet been migrated to the target database. Both updates will determine that the user has to be migrated first (see above update discussion). While the first client migrates the user first before executing the update, the second client might try to do the same (as it is executing concurrently and unaware that the migration took place in the meanwhile), experiencing a failing insert (migration) because the data has been migrated already by the first client. This scenario has to be explicitly covered by the client code. One possible approach is re-execution of the second client's update transaction.

- **Concurrent transactions with relationships**. Similar to the discussion in the previous bullet, concurrent transactions issued by clients might require migration of related data. One use case is when migrating a user, its addresses have to be migrated as well. Concurrent transactions might try to do this twice, as discussed

before. A different use case is n:1 relationships where the “1” side has to be migrated only once, while each of the “n” has to be migrated n times (each row once). In this case n-1 transactions have to be prepared for the case that the “1” side was already migrated.

The topic of validation of consistency between the source and the target comes up:

- The most helpful approach is to have distributed transactions as many possible violations would be caught automatically by it.
- Another approach is to have each transaction check if it encounters a consistent data set before it applies changes, and that it leaves a consistent data set behind after making changes. The client would abort the transaction if it encounters or would leave behind an inconsistent state (however, that would run into the larger transaction coordination problem as discussed above).
- Yet another approach is comparing source and target databases on a regular schedule. However, the question arises: what to do if inconsistencies are detected? How would those be mitigated and fixed?
- Various references (e.g. [3]) suggest constant monitoring or continued statistical analysis. While this might detect inconsistencies (or might not as it is not 100% guaranteed to find all inconsistencies), the question arises how to react to found inconsistencies and how to compensate or fix them. And, what to do if inconsistent data was read by users or downstream systems that in turn modified data based on the read data?

The references make clear that data consistency is a chief concern, but they also show that there is only a best effort, without the guarantee that all inconsistencies can be found, let alone fixed or subsequent processing that took place on inconsistent data be compensated. In such a situation, many inconsistencies might exist, and only become visible long after the switchover to the target, and maybe only when customers or users start complaining and data consistency bugs are filed.

In the *CDC database migration* variant consistency is established by process and its guarantees (like exactly once processing of transaction log records). In case one of the involved databases or systems fails, and a system failure causes inconsistency on the

target database, the mitigation is straight forward: delete the target database and start over. This is possible because the source is the primary until the switchover and no client logic depends on the target database until then. In the dual-write case a reset and restart from the beginning is possible as well but only if the source database is the primary and it is 100% consistent with the target after every transaction once the target becomes the primary. Of course, any external dependencies must be reset as well. If the target database is being used by clients while the migration is ongoing, a reset might not be possible.

The next section discusses back fill. It could have been part of the consistency discussion as well, however, due to its own complexity it is appropriate to discuss it separately.

Back fill

In the dual-write variant data is only migrated to the target database if a client accesses data that has not been migrated yet, and the client code contains migration logic that is executed. However, there is no guarantee that every data item is accessed by clients in a specific time period in which database migration is to be completed. Since data might not be accessed by clients within the given migration period, a separate process is needed to migrate these data items. This separate process is independent of the client logic in order to migrate the remaining data. This process is referred to as “back fill”.

Back fill needs to determine the data set from the source database that has not been migrated to the target database: it basically has to perform a difference between two data sets. This is a whole separate development effort. There are many ways to accomplish this and the access path management might contribute to this process. For example, [1] reports creating a database snapshot (not referring to the access path management) and using a Hadoop cluster.

No matter how back fill determines the data items that it needs to migrate from the source to the target, it has to be online as well, meaning, neither the source or target database are quiesced for this process. It needs to take into consideration:

- **Concurrent client data migration**. While back fill is ongoing, and while it determines that it needs to migrate a data item, that data item might have already been migrated concurrently or is in the process of being migrated concurrently in a

concurrent client transaction. This concurrent processing must guarantee the data consistency and back fill for this data item must fail in this case to avoid interference.

- **Data relationships**. Client code by its nature understands and is aware of data relationships (e.g., that a user has addresses in a foreign key relationship). Back fill as separate code needs to ensure that it implements the exact same data semantics in its logic. This applies not only for explicit relationships, but also for implicit relationships. Depending on how the migration code is implemented and structured, back fill might reuse the same code. If it implements its own code, migration functionality is duplicated and needs to be kept in sync with the corresponding client code.

- **Client code logic**. Some client code logic might not only migrate the data, but might update additional tables, like for example history tables to keep track of data changes. Back fill has to implement the exact same logic.

Based on the above brief discussion on back fill, it would be interesting to consider implementing the back fill process not as a separate process with its own separate logic, but find a way to have the back fill process trigger client code so that the migration of the data items it determines uses the client logic, instead of its own logic. This heavily depends on how client logic is implemented, however, the upside would be that the back fill process does not replicate logic that already exists in the client code.

None of the references I found discusses the back fill in such detail that it is clear how all the above is actually addressed and implemented.

Fallback option

In some contexts a possible fallback is prepared from the target database (the new primary) to the source database (the former primary) in case of unforeseen issues on the new primary, especially after the clients reconnected and start accessing the new primary database. In terms of data consistency this only works if the source database continues to be kept up-to-date with the primary.

In the dual-write variant this means that clients have to continue to write to both databases (the primary and source database) so that the source database is kept in sync

with the primary. All the above discussed issues remain, now in the reverse direction, so to speak.

CDC database migration. In this variant the reverse migration process is set up, from the primary to the source database. Clients operate on the new primary without any changes to their code required in order to continuously replicate the data from the primary to the source database.

Schema updates

It is possible that the client logic requires a schema update to address a change in business logic code and related data management independent of the migration logic. While it is possible to change the schema during migration, it adds complexity as not only the client code and database schema has to be updated, but the dual-write migration logic as well. Depending on the phase the migration is in, the change might be different. My recommendation would be to avoid schema updates on any database involved in the migration at all if possible during migration and defer it to after the migration completed.

A different type of schema update can be required as part of the migration logic itself. In [1] the decision was made to carry over an (existing) array of subscriptions from the source to the target schema while implementing a subscriptions table on the target schema holding the same data (basically duplicating the data for the duration of the migration: kept in the former data structure and added to a new data structure in the target database). Once the migration completed and once the subscriptions table became the primary source, the array of subscriptions was removed. This is an example of a schema change (and corresponding code change) being deferred until after the completion of the migration.

CDC database migration: in general schema changes can be detected and handled by CDC database migration; it depends on the specific technology deployed if and to what extent this is possible. However, schema changes might require to change the transformation rules transforming data from the source database before being applied to the target database. Since such a change is a transformation code change affecting the migration logic, my recommendation would be to defer schema changes to after the migration completed (if possible).

System failures

There are several systems involved in the dual-write variant:

- Client(s)
- Source database
- Target database

Each of these systems can fail independently of any of the other systems. If a client fails, then this stops the database migration. A failure has no impact on the migration, it just delays it.

If either the source database or the target database experiences and outage then the question arises how clients should proceed. Since the client code is performing the data migration and is tasked with ensuring data consistency, the only option is to stop the client code until both databases are available again. The big caveat is if the unavailable database had to be recovered from backup. If it was recovered from backup, data loss might have occurred if the backup did not capture the latest change before the outage. In this case both databases have to be synchronized first (which might be a major undertaking based on the particular context).

As a side note, an outage does not have to be a crash. It could be an unavailability as well, meaning, a restart of the database itself, or an upgrade window taking place. In general, an outage makes the database inaccessible.

CDC database migration: In the CDC database migration case, a client outage does not impact the database migration. If the source database experiences an outage, the client is impacted, but the target database is not impacted. If the source database has to be recovered from backup, it might have lost transactions. The simplest way to synchronize the target database again has two sub-cases: (a) if the target database can be restored from the same backup, then this ensures that both the source and target are in sync. (b) if the target database cannot be restored from the same backup (because for example it has a different schema), migration can start from the beginning by removing the target database and creating a new one. If the target database experiences an outage the same process takes place: if it did not experience a data loss it can simply continue. If it experienced a data loss, migration can be restarted, or replayed if the migration

technology kept the CDC records. Compared to the dual-write situation, the processes to recover in the CDC database migration case are a lot less complex, and a target database outage does not impact database migration from the viewpoint of a client impact.

Completion

The discussion so far has focused mainly on the phase before and during the migration. However, at some point the database migration is completed and cleanup has to take place.

The database migration itself is completed when the target database becomes the primary (implying that it contains the complete and consistent data set). Once the new primary is available, clients will be reconnected to the new primary and all access is executed on the new primary. Before allowing clients to access the new primary, it would be good to have a consistent backup of the database as a defined starting point.

The source database is still deployed, while not being used. While it is possible to keep it around, it will be inconsistent as soon as the first transaction commits on the new primary. Because it is inconsistent it can be turned down and the resources that it uses removed. In the case that the source database is kept in sync in order to implement the fallback option, completion of the database migration is only when the fallback option is not required anymore.

The item with the biggest impact, however, is the dual-write migration code that was added to the client's code in order to perform the database migration. This code is not required anymore and it is best practice to remove code that will not be executed anymore (and because it is a code change, testing is required). There might be the thought of keeping the code for a future database migration project. While this is certainly a consideration, it might be a better strategy to remove the code and instead consider a code refactoring after a postmortem to improve the code base instead for a future database migration.

CDC database migration. The database migration is completed when the target database becomes the primary and clients are reconnected to it. The source database can be turned down as well (unless delayed for the fallback option). However, since the client's code was never modified to perform the migration, no change, removal or refactoring is required.

Migration between tables in the same database

When migrating between tables in the same database, changes to both, the source and target table can take place in a single transaction, removing all distributed transaction concerns and the possible failure situations when having to use independent transactions for accessing source and target tables.

A particular schema related error can occur in context of foreign key relationships to the source table: when a table has a foreign key relationship (table other than source or target table) then this is a dependency that has to be addressed. At the point in the process where the new table is made the primary table (source of truth) the foreign key relationships have to be redefined in the schema definition. This is a separate step, and it might run into error situations where foreign keys do not exist (yet). Furthermore, if the primary key definition on the target table changed, the foreign key relationships have to be changed accordingly. [3] mentions possible error situations.

Non-schema based dependencies (e.g. foreign data wrappers) on the source tables might also exist. Those are not as easy to spot as dependencies that are expressed in the schema definition, however, have to be addressed as well, of course.

CDC database migration: migration between tables within a database are possible with CDC database migration as well without restriction. In this case the issue related to foreign keys (and other dependencies between tables) has to be addressed as well.

Testing

The topic of testing the migration is not discussed in detail in the references, however, testing requires a separate discussion. Testing must confirm that the database migration, once started, completes successfully. Any error during testing will have to be fixed, and testing restarted.

However, there might be edge cases that are not caught by testing immediately, and only show up when the migration of the test data occurs. In that case there are different alternatives on how to proceed:

- Stop the migration, fix the migration code, test the fixes (and update the test suite), and reset as well as restart the migration from the beginning

- Stop the migration, fix the code, fix the data, and continue the migration

The first alternative is the more reliable approach as the consistency of the data is ensured in the target database as no manual modification is required. The restart will incorporate the fix of the bug — however, this requires the ability to start the migration from the beginning.

The second alternative is fraught with possible downstream issues: this alternative relies on the ability to be able to correct the data during migration. Since fixing the code and fixing the data is concurrent to production, more errors might show up while the code is being fixed every time. This alternative works only if the target database is not accessed by clients for production, and if the clients (that are continuing to run) can deal with stopped migration. If they cannot, then this alternative does not work, instead, the migration has to continue while the code and the data is fixed.

CDC database migration: since the target database is never the primary, and since clients are not affected, the database migration can be restarted at any time from the beginning to address any issues that might come up.

Observation: data consistency uncertainty

All references clearly emphasize that data consistency is important. At the same time, all three references are not confident that the client code will produce a consistent target database from the source database. This is lack of confidence is expressed by the references describing the approaches put in place to monitor production for access issues and data issues:

- [1] expresses uncertainty about data consistency between source and target database: "*We need to be sure that it's safe to read from the new Subscriptions table: our subscription data needs to be consistent. We'll use GitHub's Scientist to help us verify our read paths.*" First of all, this means that there is the expectation that inconsistencies can occur, meaning, it is "clear" that the client code performing the migration most likely will not create a consistent target database. A separate system (Scientist) is put in place comparing the source and the target database alerting any difference when reading the "same" data item. However, it is not explained how differences are dealt with, and how the code that created the inconsistency is found and fixed. All this has

to take place during a running migration as the goal is to perform an online migration (aka, zero downtime). So while the inconsistencies are addressed, additional inconsistencies might be created. In addition, in order to find all inconsistencies, the system detecting inconsistencies must have a complete coverage of all possible inconsistencies.

- [1] also expresses that there is concern after the target database is made the primary database: "*For each code path, we'll need to use a holistic approach to ensure our changes are safe. We can't just substitute new records with old records: every piece of logic needs to be considered carefully. If we miss any cases, we might end up with data inconsistency. Thankfully, we can run more Scientist experiments to alert us to any potential inconsistencies along the way.*" This means that there is a constant and continued concern about data consistency and that the client code might not have addressed all access correctly.

- [2] realizes that code might fail and states "*Notice that in this step the "old" database is in a consistent state while the "new" database can potentially be inconsistent since the writes to it can fail while the "old" database write succeeded. It is important to let this step run for a while (several days or even weeks) before moving to the next step. This will give you the confidence that the write path of your new code works as expected and that the "new" database is configured correctly with all the replications in place.*" This means that production is monitored to detect any issues. Since there is no guarantee when errors show up, a longer time frame of observation is suggested expecting that errors show up if there are any. Like in the other references it is not described how any missing or inconsistent data is corrected.

- [3] states "*Over the course of several days, we keep an eye on our bug reporting service (Bugsnag) for incorrect single reads in production and fix them as they come in.*" This (like the other references) indicates the uncertainty if data is consistent and correct, that the bug system is able to record errors and that the bugs are fixed in production. As discussed, the bug fix probably refers to the code, and it is not clear if the bug caused downstream issues if inconsistent data was used.

- [3] employed additional monitoring to check for correctness: "*We also add a regular audit that verifies that our waivers table has 0 rows and our subscriptions table only contains processed subscriptions!*" After the data was migrated from the waivers table

(a source table), it was empty, and had to stay empty if all the code locations work properly. Since there was uncertainty if all code locations were refactored, monitoring the waiver table was necessary to see if any incorrect write takes place.

I want to highlight the fact that the references point out clearly that any code bug has to be fixed, but there is no discussion on the impact of incorrect data for downstream processing and how incorrect data is corrected. I think that in general data can be corrected only in the case of very simple data models. If dozens or hundreds of tables or collections are present, data corrections might be impossible, especially if the error rate is high.

When is dual-write online database migration an option to consider?

The above discussion points out many aspects that have to be addressed to make the dual-write variant of database migration a feasible, reliable and dependable option. Are there specific use cases where the above aspects do not apply at all or only a subset of them? Here some speculations from my side:

- A data set might represent the data as closed and complete data structure. For example, all data about a user are in a single document and there are no references to and from that document to any other. All access is by primary key only and a client reads and writes the complete document every time. No aggregation or join queries are in the system. In such a case, dual-write has less aspects to consider. What needs to be addressed are for example concurrent access supervision, system failure handling, back fill, aka, all aspects that are not related to individual query processing.

- Another use case might be an extremely simple data model and schema that allows understanding the data set completely independent of any client code accessing it. In this case it might be possible to implement verification after every database access that migrates or modifies the data set. I am not sure how many production systems fall into this category. One interesting note here is that the use case of [1] has a simple schema, yet its migration has to address a lot of the aspects discussed here.

- A very different use case is the situation where the client code can be written in such a way that it guarantees data consistency by its construction. This means that the code addresses all the above aspects completely and does not leave the possibility of data inconsistency (neither by itself, or any external event like a system failure). The hard part is to ensure that this is actually the case and to prove it.

There might be additional use cases, of course, where the dual-write variant works; the above bullet points do not represent a complete list.

Why is CDC database migration more reliable?

Interesting enough, CDC data migration falls into the category of code guaranteeing data consistency. The chief reason is that the database migration logic is independent of the client logic, aka, the migration logic is implemented by a separate migration system. It bases its functioning on the consistent behavior of the source and target database (transaction logs and transaction consistency on the query interface). If such a migration system guarantees ordered and exactly-once application of every source database change to the target database despite system failures, it produces a consistent target database.

This should not be underestimated: any error or bug that appears in the migration system can be addressed independently of the clients accessing the source database. And in general, once a bug is fixed, the migration can be restarted removing any possible inconsistent data from the target database.

What if the source database does not have a transaction log for CDC?

Not all databases support a CDC interface, meaning, a transaction log that is exactly once, transaction ordered and has a complete set of changes. This does not mean that the dual-write variant is the only option left. Instead the following can be considered: incremental batch readers.

An incremental batch reader keeps reading the source database tables using the database query interface and extracts the rows that have been modified since the last time the reader read changes. As the batch reader keeps reading, changes are extracted and can be applied to the target database.

For this to work a “flag” (aka column) has to be added to every table that stores a value for each row and that value is updated when the row is changed. Based on this an incremental reader can determine the changed rows since the last read. For example, a transaction id, or a timestamp. For example, the reader reads all rows greater than a transaction id and stores the highest transaction id read. The next time it reads again, this time from the stored transaction id.

Deletes have to be addressed as well. The two main alternatives are logical deletes that are indicated by a delete flag, or capturing deletes in a separate table.

In this case the clients have to be modified in order to store an additional value in the additional column (an additional column for each table). While it requires to modify the client code, the logic is a lot simpler and less brittle than the dual-write variant since it updates only values related to the regular data model without having to implement the migration logic. On the target database side, clients can ignore the additional column by leaving those out in the query projection clause.

Conclusion

The above discussion compared two variants of online database migration and outlined the many risks when using the dual-write variant compared to the CDC based variant. The discussed issues and risks are not a 100% complete list, depending on the particular migration additional issues and risks might have to be addressed.

My recommendation is: when embarking on online database migration, before choosing the dual-write variant, analyze the CDC variant carefully first in order to base the decision on the effort, the risks, and any error situations that might come up and have to be addressed. Above all, data integrity and consistency must be the guiding principle unless that does not play a role at all in your context.

My conclusion is to not consider the dual-write variant as a regular approach to database migration, but as an edge-case approach and only if all other options were invalidated or ruled out. And even then I would attempt to ensure data consistency by code design, and not by monitoring and fixing migration bugs in the production system.

References

- [1] Online migrations at scale
- [2] Safe Database Migration Pattern Without Downtime (Internet archive link)
- [3] Zero Downtime Table Migrations using a Double Write Methodology
- [4] https://www.quora.com/How-big-companies-migrate-from-one-database-to-another-without-losing-data-i-e-database-independent/answer/Siddharth-Anand
- [5] https://cloud.google.com/solutions/database-migration-concepts-principles-part-1
- [6] https://cloud.google.com/solutions/database-migration-concepts-principles-part-2
- [7] https://www.slideshare.net/aviranwix/road-to-continuous-delivery-wixcom

Disclaimer

Christoph Bussler is a Solutions Architect at Google, Inc. (Google Cloud). The opinions stated here are my own, not those of Google, Inc.

Database Migration | Change Data Capture | Database