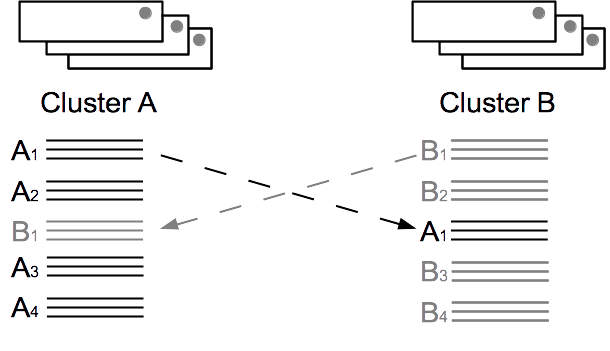
Understanding Conflict Resolution

One aspect of database replication that is unique to cross datacenter replication (XDCR) is the need to prepare for and manage conflicts between the databases. Conflict resolution is not an issue for passive replication since changes travel in only one direction. However, with XDCR it is possible for changes to be made to the same data at approximately the same time on two databases. Those changes are then sent to the other database, resulting in possible inconsistencies or invalid transactions.

For example, say clusters A and B are processing transactions as shown in [Figure 11.8, “Transaction Order and Conflict Resolution”](https://docs.voltdb.com/UsingVoltDB/DbRepHowToActive.php#DbRepConflictFig). Cluster A executes a transaction that modifies a specific record and this transaction is included in the binary log A1. By the time cluster B receives the binary log and processes A1, cluster B has already processed its own transactions B1 and B2. Those transactions may have modified the same record as the transaction in A1, or another record that would conflict with the change in A1, such as a matching unique index entry.

**Figure 11.8. Transaction Order and Conflict Resolution**



Under these conditions, cluster B cannot simply apply the changes in A1 because doing so could violate the uniqueness constraints of the schema and, more importantly, is likely to result in the content of the two database clusters diverging. Instead, cluster B must decide which change takes priority. That is, what resolution to the conflict is most likely to produce meaningful results or match the *intent* of the business application. This decision-making process is called *conflict resolution*.

No matter what the resolution, it is important that the database administrators are notified of the conflict, why it occurred, and what action was taken. The following sections explain:

* How to avoid conflicts
* How VoltDB resolves conflicts when they do occur
* What types of conflicts can occur
* How those conflicts are reported

#### 

#### **Designing Your Application to Avoid Conflicts**

VoltDB uses well-defined rules for resolving conflicts. However, the best protection against conflicts and the problems they can cause is to design your application to avoid conflicts in the first place. There are at least two things you can do in your client applications to avoid conflicts:

* **Use Primary Keys**

It is best, wherever possible, to define a primary key for all DR tables. The primary key index greatly improves performance for finding the matching row to apply the change on a consumer cluster. It is also required if you want conflicts to be resolved using the standard rules described in the [following section](https://docs.voltdb.com/UsingVoltDB/DbRepHowToActive.php#DbRepConflictProcess). Any conflicting action without a primary key is rejected.

* **Apply related transactions to the same cluster**

Another tactic for avoiding conflicts is to make sure any autonomous set of transactions affecting a set of rows are all applied on the same cluster. For example, ensuring that all transactions for a single user session, or associated with a particular purchase order, are directed to the same cluster.

#### **How Conflicts are Resolved**

Even with the best application design possible, errors in program logic or operation may occur that result in conflicting records being written to two or more databases. When a conflict does occur, VoltDB follows specific rules for resolving the issue. The conflict resolution rules are:

* Conflicts are resolved on a per action basis. That is, resolution rules apply to the individual INSERT, UPDATE, or DELETE operation on a specific tuple. Resolutions are not applied to the transaction as a whole.
* The resolution is that the incoming action is accepted (that is, applied to the receiving database) or rejected.
* Only actions involving a table with a primary key can be accepted, all other conflicting actions are rejected.
* Accepted actions are applied as a whole — the entire record is changed to match the result on the producer cluster. That means for UPDATE actions, all columns are written not just the columns specified in the SQL statement.
* For tables with primary keys, the rules for which transaction wins are, in order:
  1. DELETE transactions always win
  2. If neither action is a DELETE, the last transaction (based on the timestamp) wins

Let's look at a simple example to see how these rules work. Assume that the database stores user records, using a numeric user ID as the primary key and containing columns for the user's name and password. A user logs on simultaneously in two locations and performs two separate updates: one on cluster A changing their name and one on cluster B changing the password. These updates are almost simultaneous. However, cluster A timestamps its transaction as occurring at 10:15.00.003 and cluster B timestamps its transaction at 10:15.00.001

The binary logs from the two transactions include the type of action, the contents of the record before and after the change, and the timestamps — both of the last previous transaction and the timestamp of the new transaction. (Note that the timestamp includes both the time and the cluster ID where the transaction occurred.) So the two binary logs might look like the following.

**Binary Log A1:**

|  |  |
| --- | --- |
| Action: UPDATE Current Timestamp:  1, 10:15.00.003 Previous Timestamp: 1, 06:30.00.000 | |
| Before  UserID: 12345  Name: Joe Smith  Password: abalone | After  UserID: 12345  Name: **Joseph Smith**  Password: abalone |

**Binary Log B1:**

|  |  |
| --- | --- |
| Action: UPDATE Current Timestamp:  2, 10:15.00.001 Previous Timestamp: 1, 06:30.00.000 | |
| Before  UserID: 12345  Name: Joe Smith  Password: abalone | After  UserID: 12345  Name: Joe Smith  Password: **flounder** |

When the binary log A1 arrives at cluster B, the DR process performs the following steps:

1. Uses the primary key (12345) to look up the current record in the database.
2. Compares the current timestamp in the database with the previous timestamp in the binary log.
3. Because the transaction in B1 has already been applied on cluster B, the time stamps do not match. A conflict is recognized.
4. A primary key exists, so cluster B attempts to resolve the conflict by comparing the new timestamp, 10:15.00.003, to the current timestamp, 10:15.00.001.
5. Because the new timestamp is the later of the two, the new transaction "wins" and the change is applied to the database.
6. Finally, the conflict and resolution is logged. (See [Section 11.3.8.4, “Reporting Conflicts”](https://docs.voltdb.com/UsingVoltDB/DbRepHowToActive.php#DbRepConflictLog) for more information about how conflicts are reported.)

Note that when the UPDATE from A1 is applied, the change to the password in B1 is overwritten and the password is reset to "abalone". Which at first looks like a problem. However, when the binary log B1 arrives at cluster A, the same steps are followed. But when cluster A reaches steps #4 and 5, it finds that the new timestamp from B1 is older than the current timestamp, and so the action is rejected and the record is left unchanged. As a result both databases end up with the same value for the record. Essentially, the password change is dropped.

If the transaction on cluster B had been to delete the user record rather than change the password, then the outcome would be different, but still consistent. In that case, when binary log A1 reaches cluster B, it would not be able to find the matching record in step #1. This is recognized as a DELETE action having occurred. Since DELETE always wins, the incoming UPDATE is rejected. Similarly, when binary log B1 reaches cluster A, the previous timestamps do not match but, even though the incoming action in B1 has an older timestamp than the UPDATE action in A1, B1 "wins" because it is a delete action and the record is deleted from cluster A. Again, the result is consistent across the two databases.

The real problem with conflicts is when there is no primary key on the database table. Primary keys uniquely identify a record. Without a primary key, there is no way for VoltDB to tell, even if there are one or more unique indexes on the table, whether two records are the same record modified or two different records with the same unique key values.

As a result, if there is a conflict between two transactions without a primary key, VoltDB has no way to resolve the conflict and simply rejects the incoming action. Going back to our example, if the user table had a unique index on the user ID rather than a primary key, and both cluster A and cluster B update the user record at approximately the same time, when binary log A1 arrives at cluster B, it would look for the record based on all columns in the record and fail to find a match.

However, when it attempts to insert the record, it will encounter a constraint violation on the unique index. Again, since there is no primary key, VoltDB cannot resolve the conflict and rejects the incoming action, leaving the record with the changed password. On cluster A, the same process occurs and the password change in B1 gets rejected, leaving cluster A with a changed name column and database B with a changed password column — the databases diverge.

#### **What Types of Conflict Can Occur**

The preceding section uses a simple case of conflicting UPDATE transactions to illustrate the steps involved in conflict resolution. However, there are several different types of conflict that can occur. First, there are three possible actions that the binary log can contain: INSERT, UPDATE, or DELETE. There are also three types of conflicts that can be generated:

* **Missing row** — The affected row is missing from the consumer database.
* **Timestamp mismatch** — The affected row exists in the consumer database, but has a different timestamp than expected (in other words, it has been modified).
* **Constraint violation** — Applying the incoming action would result in one or more constraint violations on unique indexes.

A missing row means that the binary log contains an UPDATE or DELETE action, but the affected row cannot be found in the consumer database. (A missing row conflict cannot occur for INSERT actions, since INSERT assumes no such row exists.) In the case of a missing row conflict, VoltDB assumes a DELETE action has removed the affected row. Since the rule is that DELETE wins, this means the incoming action is rejected.

Note that if the table does not have a primary key, the assumption that a DELETE action removed the row is not guaranteed to be true, since it is possible an UPDATE changed the row. Without a primary key, there is no way for the DR process to find the matching row when some columns may have changed, so it assumes it was deleted. As a result, an UPDATE could occur on one cluster and a DELETE on the other. This is why assigning primary keys is recommended for DR tables when using XDCR.

If the matching primary key is found, it is still possible that the contents of the row have been changed. In which case, the timestamps will not match and a timestamp mismatch conflict occurs. Again, this can happen for UPDATE and DELETE actions where an existing row is being modified. If the incoming action is a DELETE, it takes precedence and the row is deleted. If not, if the incoming action has the later of the two timestamps, it is accepted. If the existing record has the later timestamp, the incoming action is rejected.

Finally, whether the timestamps match or not, with an INSERT or UPDATE action, it is possible that applying the action would violate one of more unique index constraints. This can happen because another row has been updated with matching values for the unique index or another record has been inserted with similar values. Whatever the cause, VoltDB cannot apply the incoming action so it is rejected. Note that for a single action there can be more than one unique index that applies to the table, so there can be multiple constraint violations as well as a possible incorrect timestamp. When a conflict occurs, all conflicts associated with the action are included in the conflict log.

To summarize, the following chart shows the conflicts that can occur with each type of action and the result for tables with a primary key.

| **Action** | **Possible Conflict** | **Result for Tables with Primary Key** |
| --- | --- | --- |
| INSERT | Constraint violation | Rejected |
| UPDATE | Missing row Timestamp mismatch Constraint violation | Rejected Last transaction wins Rejected |
| DELETE | Missing row Timestamp mismatch | Accepted (no op) Accepted |

Note: - Conflict happens only when we perform Asynchronous update to replica. If we update is synchronously using 2-Phase Commit taking lock on all the replicas conflict will not happens

<https://docs.voltdb.com/UsingVoltDB/DbRepHowToActive.php#DbRepConflictProcess>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*