Concurrency Control

Introduction

ClustrixDB uses a combination of Multi-Version Concurrency Control (MVCC) and 2 Phase Locking (2PL) to support mixed read-write workloads. In our system, readers enjoy lock-free snapshot isolation while writers use 2PL to manage conflict. The combination of concurrency controls means that readers never interfere with writers (or vice-versa), and writers use explicit locking to order updates.

Multi-Version Concurrency Control

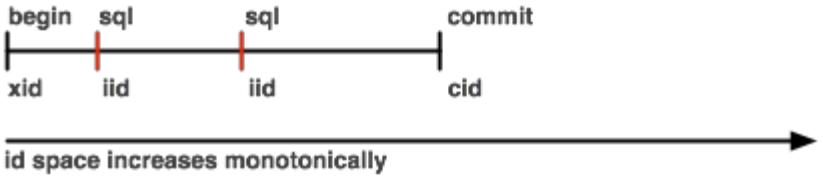
ClustrixDB implements a distributed MVCC scheme to ensure that readers are lockless, so that readers and writers never interfere with each other. As writers modify rows within the system, ClustrixDB maintains a version history of each row. Each statement within a transaction uses lock-free access to the data to retrieve the relevant version of the row.

Visibility Rules

Visibility rules within ClustrixDB are governed by sets of ids (identifiers) associated with each transaction and statement execution. Rows modified by a transaction will only become visible to other transactions after the modifying transaction commits. Once the transaction commits, it generates a commit id (cid) at which the modification becomes visible.

The following chart displays the transaction lifespan.

Transaction Lifespan



xid	Transaction start id. Marks the logical beginning of the transaction.
iid	Invocation id. Marks the beginning of a statement within a transaction.
cid	Commit id. Marks the id at which transaction changes are visible to other transactions.

The following table describes MVCC visibility rules in different isolation levels. Note that ClustrixDB does not support the read uncommitted level.

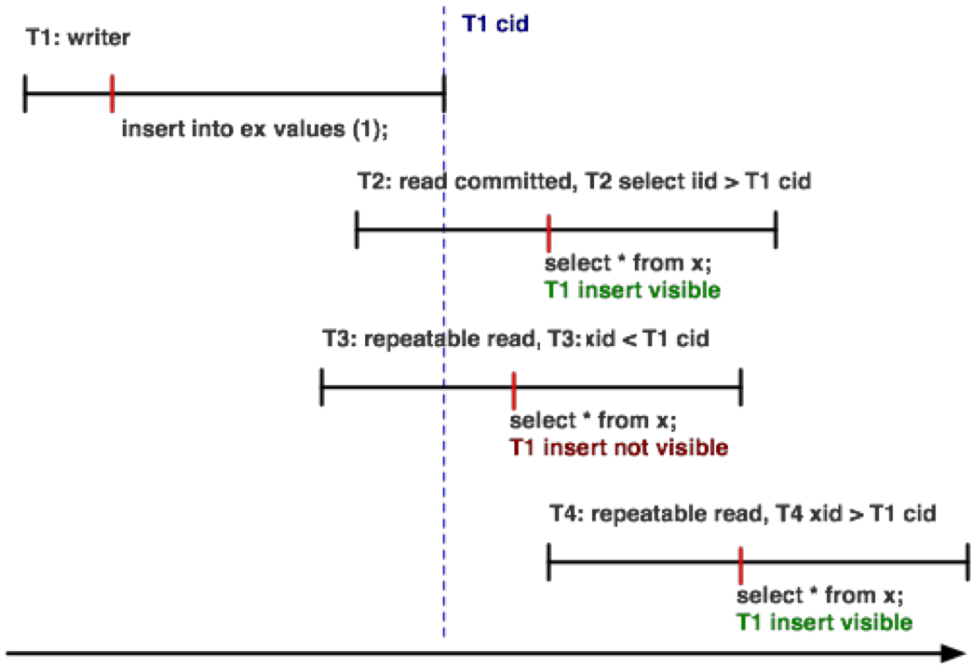
Isolation Levels

Each isolation level has a different set of visibility rules. The following table describes the basic rules for row visibility between transactions.

Isolation Level	Snapshot Anchor	Comment
Read committed	statement	More strict than read committed as defined by ANSI. Allows a per-statement consistent snapshot read. Each subsequent statement in a transaction gets a new iid, so every new statement can see rows committed before statement started executing (but never during). Most similar to Oracle's consistent read isolation. Rows are visible when statement (invocation) id > the modifying transaction commit id.

Repeatable read (default)	transaction	More strict than repeatable read as defined by ANSI. Allows a per-transaction consistent snapshot read. Each subsequent statement in a transaction sees the database at the start of transaction. Transactions may also observe changes to the database made within the transaction. Rows are visible when transaction id > the modifying transaction commit id.
Serializable	transaction	Strict ANSI isolation level. Used by the system to perform data moves within the cluster. Rows visible when transaction id > modifying transaction commit id. Database returns an error when the MVCC scheduler cannot guarantee transaction serializability. **Note: serializable isolation not currently available to end user transactions.**

The following example demonstrates how transaction visibility works at different isolation levels.

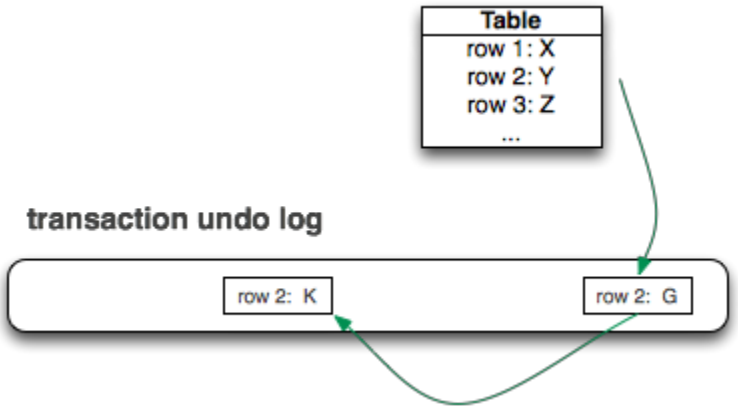


ID Generator

ClustrixDB uses a globally unique ordered transaction id (xid) along with statement invocation id (iid) to control row visibility. Both the xid and iid generators use a combination of system wide clock and a unique node id to create a globally unique ordered identifier.

Version History and Garbage Collection

ClustrixDB maintains version history through the undo log. Since the undo log must already maintain information about the previous version of the row for transaction rollback, we can rely on using this information to access previous versions of the row for MVCC. The technique allows ClustrixDB to maintain tables clustered by primary key instead of managing a large heap space. For inserts and updates, garbage collection occurs when we trim the undo log. However, ClustrixDB keeps enough undo log history to service currently executing transactions. In addition to limiting undo log trim to local checkpoint rules, we also limit trimming based on the id of the oldest transaction within the system.



The diagram above demonstrates how the system keeps multiple versions of the row in the undo log. Each row contains a Log Sequence Number (LSN) which points at the previous version of the row. We know that we've reached the end of the history chain when the previous LSN pointer precedes the trim LSN.

Two Phase Locking for Writes

Optimistic concurrency controls do not work well in the presence of conflict (two transactions attempting to update the same row simultaneously). In such cases, a purely MVCC system would roll back one or both of the conflicting transactions and restart the operation. Since ClustrixDB does not require the use of predetermined transactions (e.g. all logic within a stored procedure), such errors could bubble up to the application. Additionally, it's possible to create live-lock scenarios where transactions cannot make progress because of constant conflict.

To overcome these issues ClustrixDB uses locking for writer-writer conflict resolution. Writers always read the latest committed information and acquire locks before making any changes.

Distributed Lock Manager

ClustrixDB implements a distributed lock manager to scale write access to hot tables. Within the cluster, each node maintains a portion of the lock domain. No single node holds all of the lock information for the cluster.

Row Level and Table Level Locking

ClustrixDB implements row level locks for transactions which touch a few rows at a time (a runtime configurable variable). For statements which affect a significant portion of a table, the query optimizer will promote row level locks to a table lock.