In-Memory Tables

- What is an In-Memory Table?
- Why In-Memory Tables?
- How do In-Memory Tables Work?
- Summary of In-Memory Table Features
- Caveats and Limitations of In-Memory Tables

What is an In-Memory Table?

ClustrixDB's In-Memory tables are kept in memory and are not written to persistent storage. They are not durable. In the event of power failure or controlled cluster shutdown, In-Memory tables will be lost.

In-Memory tables can be used in conjunction with persistent tables for queries and multi-statement transactions. They are designed for atomicity, consistency, and isolation, but not durability (hence they are not fully ACID compliant). In-Memory tables can be moved to persistent storage (and made durable) by using CREATE TABLE as SELECT FROM, Backup, Restore, or ALTER.

As with persistent tables, In-Memory tables are accessed via a standard SQL interface. ClustrixDB stores multiple copies of In-Memory tables across the cluster to allow for greater concurrency and to make them fault-tolerant. If you lose a single replica, ClustrixDB will reprotect the In-Memory tables the same as persistent tables. In-Memory tables are sliced and rebalanced across the cluster to ensure an even distribution of memory and processing. Queries for In-Memory tables use the same query planner and are executed the same as for other types of tables. You can use all of the standard ClustrixDB utilities on In-Memory tables such as: Backup, Restore, clustrix_import, etc.

Why In-Memory Tables?

In-Memory tables provide performance advantages for tables that meet these criteria:

- The data does not need to be durable and if the data is lost, it can re-loaded or reconstructed without serious impact to the application.
- Table fits in memory and does not grow aggressively.
- Table has a high write to read ratio - i.e. data loading or frequent updates.
- Table uses primarily auto-committed, standalone transactions versus multi-statement transactions.
- Tables used by applications with mixed durability requirements are also potential In-Memory candidates. For example, if a table could use RELAX ED durability, but the remainder of an application cannot, placing just that one table in memory can potentially improve performance while still ensuring the durability of the remaining application data.

Our internal testing indicates that data loading (both bulk loads and single-row data ingestion) is significantly faster into an In-Memory table versus a persistent table.

How do In-Memory Tables Work?

Differences from Tables Stored on Disk

In-Memory tables:

- Are stored non-durably in memory whereas persistent tables are stored on disk.
- Are limited to a row size of 32k (enforced at INSERT/UPDATE). Maximum row size for tables stored on persistent storage is 64MB.
- Include a memory overhead of 312 bytes per row, whereas the corresponding row overhead for persistent tables is much lower and is stored on disk.
- Use Optimistic MVCC, which means that colliding transactions will need to be reprocessed by your application (i.e. retried, rejected, other). Persistent tables use ClustrixDB's standard MVCC processing to facilitate Concurrency Control.
- Queries to In-memory tables do not take out locks. For example, queries that perform a SELECT ... FOR UPDATE will not block

DDL Operations for In-Memory Tables

CREATE TABLE

ClustrixDB stores data in three types of containers. The container's type is established when a table is created. The allowable containers types are:

- **layered** Data is stored persistently on disk in layered B-trees.
- **btree** Data is stored persistently on disk in B-trees.
- **skiplist** Data is stored in memory in skip lists.

Specify container = skiplist to create an In-Memory table (quotes are optional).

```
SQL> CREATE TABLE sample
       (Id integer primary key,
        type varchar(5),
        description varchar(100)
       ) container=skiplist;
```

ClustrixDB also supports the MySQL syntax of “ENGINE=MEMORY” to create an In-Memory table.

ALTER TABLE

Tables may be moved from in-memory to persistent and vice versa by altering the table’s container type. This also modifies index storage.

Make an In-Memory table persistent and durable

```
SQL> ALTER TABLE sample container=layered;
```

Convert a persistent table to an In-Memory table (non-durable)

```
SQL> ALTER TABLE sample container=skiplist;
```

ClustrixDB also supports the MySQL syntax of “ENGINE=MEMORY”.

As with other types of Online Schema changes, altering an In-Memory table to be persistent (and vice versa) does not block reads or writes.

You may also DROP and TRUNCATE In-Memory tables. They can have triggers associated with them, be used in stored procedures, views, and joined with persistent tables. Online schema changes are allowed for In-Memory tables. For information on partitioning In-Memory tables, please see Partitioned Tables.

Fault Tolerance and Durability of In-Memory Tables

If a single node fails, In-Memory tables will continue to be operational and the ClustrixDB Rebalancer will work to make sure there are sufficient replicas of all tables.

In the event of a power failure or cluster restart, data stored in In-Memory tables will be lost, along with a portion of that table's metadata. The table’s definition is preserved on disk, but the data, along with metadata for slices, replicas, and indices, are lost. Before any queries can be run against an In-Memory table, including a query to populate it, all metadata for the table must be restored.

If you query the table before this metadata exists, you will see this error:

```
SQL> select * from sample;
SQL> ERROR 1 (HY000): [6144] No distribution for representation: No final distribution for representation
"__idx_sample__PRIMARY"
```

To restore a table’s metadata, perform one of the following:

Option 1: Drop the In-Memory table, then restore it from the most recent backup. See ClustrixDB Fast Backup and Restore.

Option 2: Combine the DROP and CREATE TABLE to rebuild the table definition.

```
SQL> DROP TABLE sample;
SQL> CREATE TABLE sample
          (Id integer primary key,
           type varchar(5),
           description varchar(100)
          ) container=skiplist;
```

After re-creating or repairing your In-Memory table’s definition, you must reload your data in its entirety.

Optimistic MVCC

In-Memory tables use Optimistic MVCC. Persistent tables use ClustrixDB’s standard Multi-Version Concurrency Control (MVCC). The two methods are more similar than dissimilar.

Note that all database writes first perform a read followed by the insert, update, or delete, so each write operation is actually two steps.

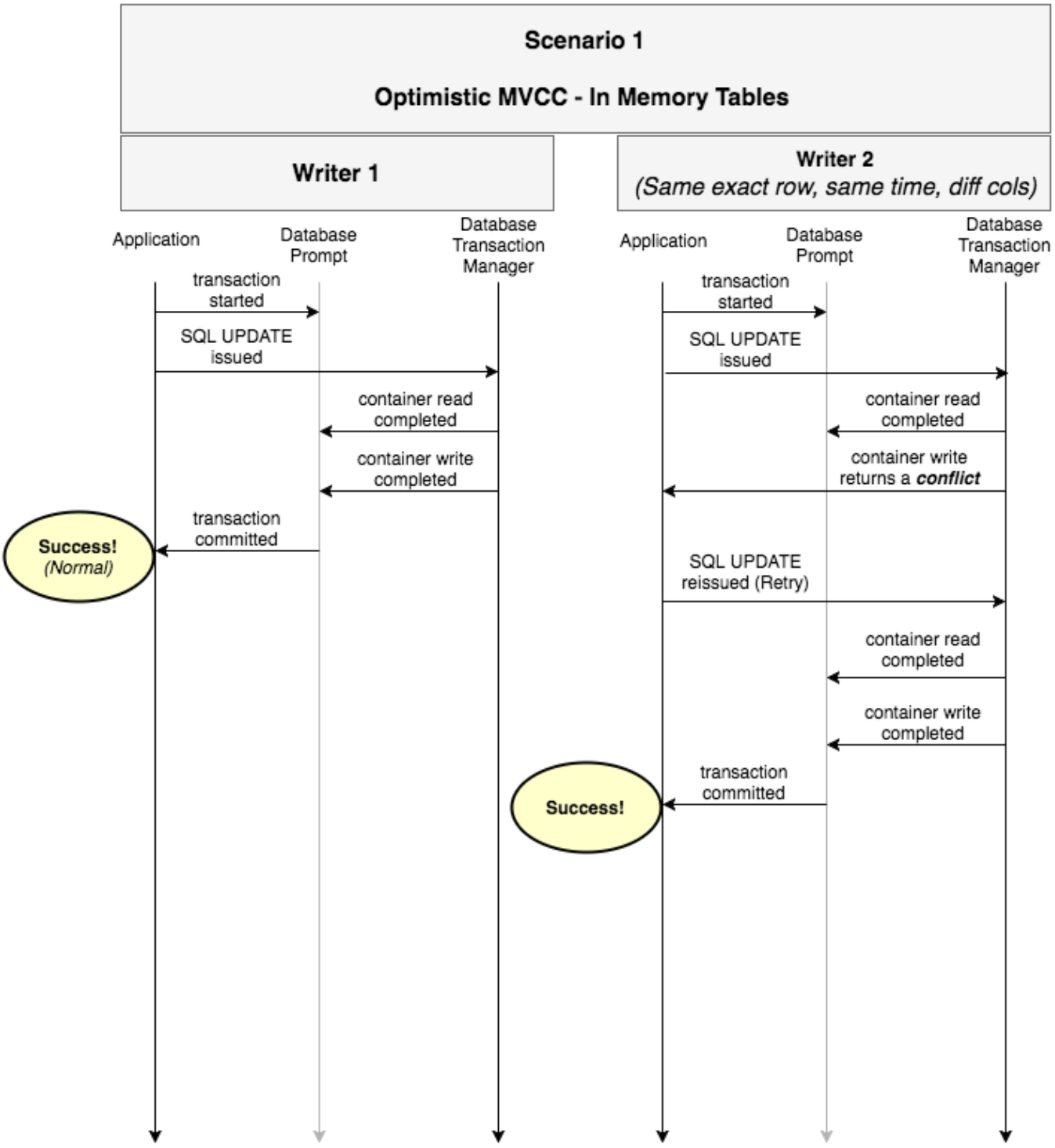
The primary difference is that Optimistic MVCC does not lock table rows to prevent contention, but instead gracefully rejects colliding statements. With Optimistic MVCC, a statement within a transaction will be rejected if it attempts to write the same keyed row at the exact same time as another write transaction.

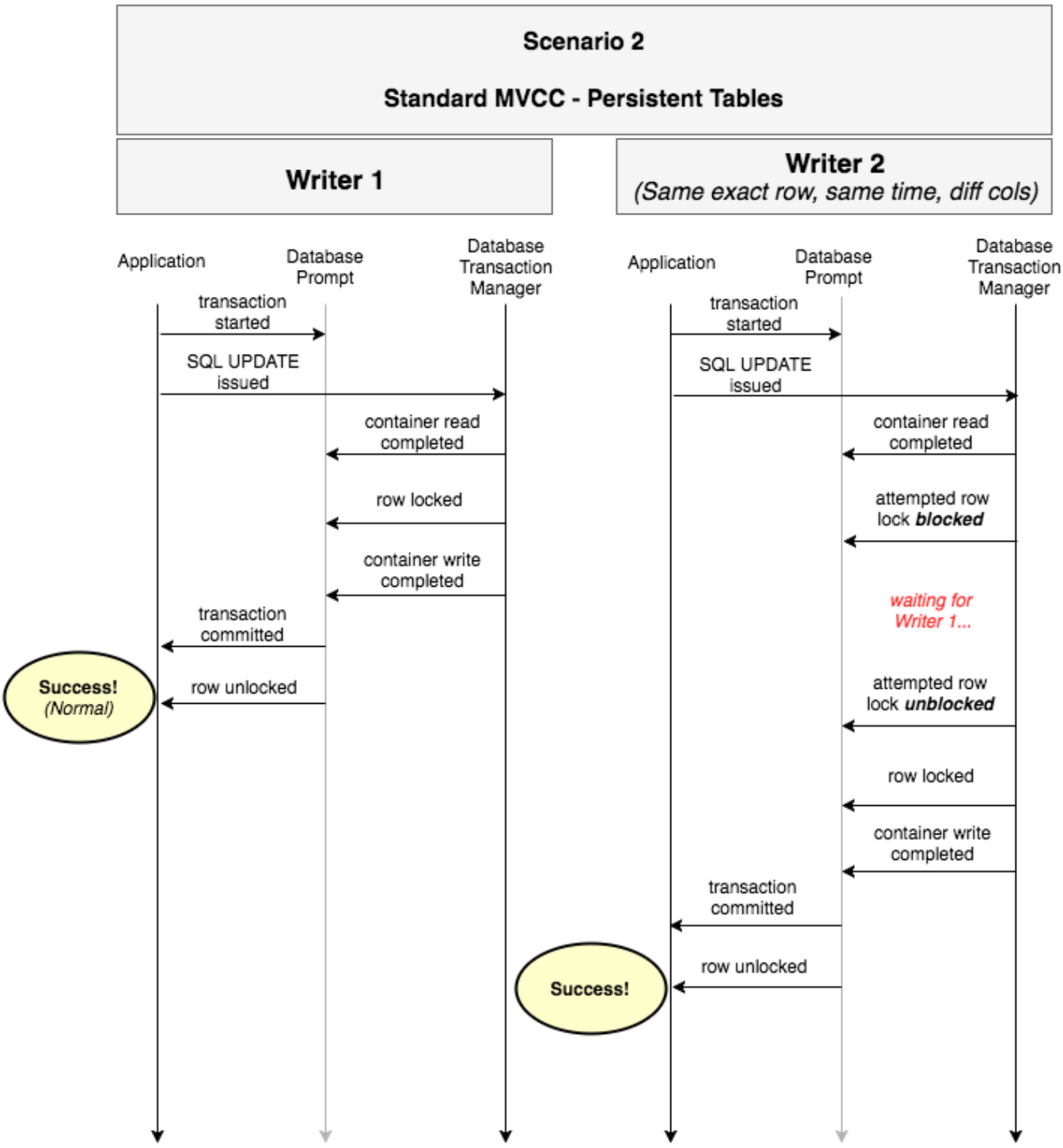
In-house tests on a small data set with no coordination and lots of concurrency produced collisions 2% - 5% of the time. Depending on your use case, these errors can be auto re-tried by your application. (Setting the global variable autocommit to TRUE will cause a failed statement within a transaction to be automatically retried.)

The possible errors are:

> *ERROR 1 (HY000): [5123] Container optimistic MVCC conflict:*
>
> *ERROR 1062 (23000): [5120] Duplicate key in container:*

To illustrate a case when those errors could arise, let’s compare Optimistic MVCC to standard MVCC. This sample shows two concurrent writers accessing different columns of the same row at the exact same time (rare). The left-hand diagram shows processing for Optimistic MVCC, as used for In-Memory Tables. The right-hand diagram shows similar processing using ClustrixDB's Standard MVCC processing, as used for Persistent Tables.


Scenario 1
Optimistic MVCC - In Memory Tables
Writer 1
Writer 2
(Same exact row, same time, diff cols)
Application
Database Prompt
Database Transaction Manager
transaction started
SQL UPDATE issued
container read completed
container write completed
transaction committed
Success!
(Normal)
container write returns a conflict
SQL UPDATE reissued (Retry)
Success!



Garbage Collection for In-Memory Tables

ClustrixDB maintains version history in undo logs for all table types. For In-Memory tables, this undo information is stored in a separate log file in memory. As with persistent tables, entries are purged from this log once no transactions reference them. You may observe that long-running transactions (i.e. ALTE R TABLE or TRUNCATE TABLE) consume memory until they complete.

Rebalancer Details

As with persistent data, the ClustrixDB Rebalancer detects imbalances in data distribution and load for In-Memory tables and will redistribute data for those tables with a background process. If a node is lost, the Rebalancer will also reprotect In-Memory tables to ensure that the requisite number of replicas are maintained.

Global Variables for Memory Management

ClustrixDB provides the following controls for managing the amount of memory usable by In-Memory tables:

Variable	Description	Default
max_memory_table_limit_mb	Maximum memory usable for In-Memory tables per node (in MB). This memory is not reserved or pre-allocated.	16

These global variables control how memory limits are enforced and when alerts are sent if those limits are exceeded. For information on ClustrixDB's Alerter, please see Database Alerts.

Variable	Description	Default
memory_table_system_full_error_percentage	Fail system queries when space usage for in-memory tables surpasses this percentage.	97
memory_table_system_full_warn_percentage	Warn about system queries when space usage for in-memory tables surpasses this percentage.	95
memory_table_user_full_error_percentage	Fail user queries when space usage for in-memory tables surpasses this percentage.	90
memory_table_user_full_warn_percentage	Warn about user queries when space usage for in-memory tables surpasses this percentage.	80

The Health Dashboard of the ClustrixGUI Administrative Tool also displays information about your cluster's memory utilization.

Summary of In-Memory Table Features

The following chart summarizes the differences between In-Memory and persistent tables.

Physical Properties	In-Memory	Persistent Storage Comparison
Storage	In-Memory (exclusively)	On disk (exclusively)
Row Size Limit	32K (enforced on INSERT/UPDATE)	64MB (enforced on INSERT/UPDATE)
Container Structure (data and indices)	skiplist	layered, btree
ClustrixDB Compatibility	**In-Memory**	**Persistent Storage Comparison**
Standard SQL DDL and DML Interface	Yes	Yes
MVCC	Optimistic MVCC	MVCC
Row Locking/Latches	No	Yes
Sliced	Yes	Yes
Replicated	Yes	Yes
Distributed/Balanced Cluster-wide	Yes	Yes
Fault Tolerant (if you lose a node, ClustrixDB reprotects)	No	Yes
Join In-Memory tables and persistent tables	Yes	Yes
Foreign Key Support	Yes	Yes
Utilizes Sierra Query Planner and Execution	Yes	Yes
Views	Yes	Yes
Indexes	Yes	Yes
Triggers	Yes	Yes
Stored Procedures	Yes	Yes
Partitioned Tables	Yes	Yes
Temporary Tables	Yes	Yes
ACID Compliance	**In-Memory**	**Persistent Storage Comparison**

Atomic, Consistent, and Isolated transactions cluster-wide	Yes	Yes
Durable data and transactions	No	Yes
Utilities/Other	**In-Memory**	**Persistent Storage Comparison**
clustrix_import	Yes	Yes
LOAD DATA INFILE	Yes	Yes
Backup and Restore	Yes	Yes
mysqldump	Yes	Yes
Online Schema Changes	Yes	Yes

Caveats and Limitations of In-Memory Tables

When using In-Memory tables, please keep the following limitations and caveats in mind:

- Be sure your cluster's available memory fully accommodates your planned In-Memory table(s).
- Since In-Memory tables are not durable, recovery processes such as backup/restore or reloading should be planned and include REPAIR table and index recreation.
- A strategy to address Optimistic MVCC collisions should be developed. ClustrixDB will automatically retry a conflicting transaction if autocommit is TRUE.
- In-Memory tables have a maximum row size of 32K that is imposed upon insert or update.
- There is an overhead of 312 bytes of memory per row. Depending on the width of each table row, this additional overhead can significantly affect the total space needed for an In-Memory table. For example, if your average row size is 100 bytes, loading that same table In-Memory will require approximately four times more space, i.e. 400+ bytes per row vs. 100.
- The maximum amount of memory that can be allocated to In-Memory tables is up to 50% of available memory.