HOW CLUSTRIXDB RDBMS SCALES WRITES & READS

Distributed Relational Database Architecture: An Overview

Scaling out a SQL RDBMS while maintaining ACID guarantees in realtime is a very large challenge. Most scaling DBMS solutions relinquish one or many realtime transactionality requirements. ClustrixDB achieves near-linear scaling of both write and read queries with full ACID compliance, by a combination of 'bringing the query to the data' and automatically distributing the data across all nodes in the cluster. Read this white paper for more information including how ClustrixDB leverages a Cascades Planner, MVCC, 2 Phase Locking, and the Paxos consensus protocol.

ClustrixDB WHITE PAPER

The Relational Database Scalability Problem

Traditional relational databases were not designed for massive scale. Despite decades of effort, extensions and workarounds, this fundamental limit remains. The problem is architectural and must ultimately be addressed with a new elastic RDBMS design that scales without compromising consistency.

Some DBAs argue that there are, in fact, relational DBMS (RDBMS) scaling techniques: bigger more powerful servers ("scale-up"), read slaves, master-master replication, sharding, distributed partitions, in-memory, and even cloud based RDBMS as a service or DBaaS. Each of these techniques works around some portion of the RDBMS scalability problem. All add significant complexity, cost, and/or fragility while failing to completely address the core issue. Truly scaling an RDBMS requires scaling both writes and reads without compromising performance, ACID guarantees, referential integrity, and supportability.

SCALE UP: A LIMITED SOLUTION

RDBMS scale-up is inherently limited by hardware architecture: processing, memory, IO, networking, storage. Implementing bigger (and more expensive) servers offers temporary relief at best. Scale-up is a problem for several reasons:

- **Manually labor intensive, = > $$$ to implement, with painful tech refresh, = > cost and downtime.**
- **Increasingly expensive.**
- **Requires buying bigger, = > $$$ server, storage, and network infrastructure upfront than needed.**
- **Dependent on Moore's law, even though Moore's law has noticeably slowed and may be hitting a wall.**
- **Hoping that the Relational growth will slow or stop before hardware limits or next generation is available is a recipe for anxiety, frustration, and failure.**

A truly scalable RDBMS architecture meets the following criteria:

- Scales-out (a.k.a. horizontal scaling). The addition of either physical or virtual servers (nodes) extends the RDBMS instance. Each node should increase the RDBMS's ability to scale both writes and reads linearly.
- Scaling and maintenance are Non-Disruptive. Adding hardware resources, tech refreshes, software updates, microcode upgrades, etc. have no material affect on the RDBMS or applications.
- Ensures high availability and/or fault tolerance. No single point of failure (No SPOF). Failure of any component should not cause application downtime.
- Scaling is Cost-Effective. Few IT organizations, cloud operations, or managed service providers have unlimited budgets. Scaling should be pay-as-you-go, meaning you only pay for what is required when it is required.

Table 1: Comparison of RDBMS Scaling Techniques

	Scale-Up	Read Slaves	Master-Master	Sharding	Distributed Partition	In-Memory	DBaaS	ClustrixDB
Scales Writes	Yes[1]	No	Partially[2]	Yes	Yes[3]	DRAM limited	Yes[1]	Yes
Scales Reads	Yes[1]	Yes[3]	Yes[3]	Yes	Yes[3]	DRAM limited	Yes[3]	Yes
Scale-out	No	No	Partially[3]	Yes	No	DRAM limited	No	Yes
DBMS & App Non-Disruptive	Yes[1]	Yes[3]	No[3]	No	No	Potentially[3]	No	Yes
No SPOF / Fault Tolerance	Server HW dependent	Yes[3]	Yes[3]	No	No	Server HW dependent	Yes	Yes
Cost Effective	No[1]	No[4]	No[2]	No	No	No[2]	No[2]	Yes

[1] Scale-up continues to scale until the biggest single server is provisioned. Then you 'hit the wall' and cannot scale-up further.
[2] Master-Master adds HA, but cannot give linear write-scale
[3] Potential Consistency issue (read-slave), and/or conflict resolution code needed (master-master)
[4] Each new server does not scale-out reads linearly.

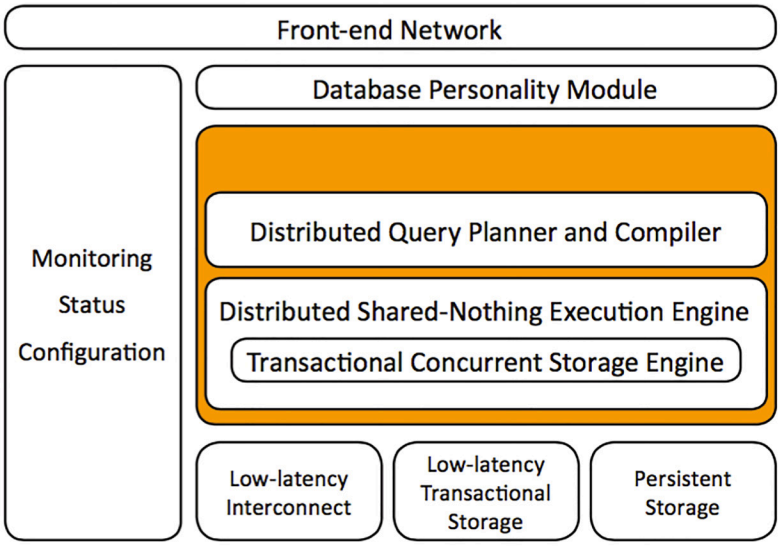
ClustrixDB: Designed from the Ground Up to Scale

ClustrixDB is a unique shared-nothing clustered scale-out relational database. It solves the RDBMS scalability problem through the extensive use of systemic parallelism.

The following describes an overview of how ClustrixDB's architecture accomplishes near-linear scale-out.

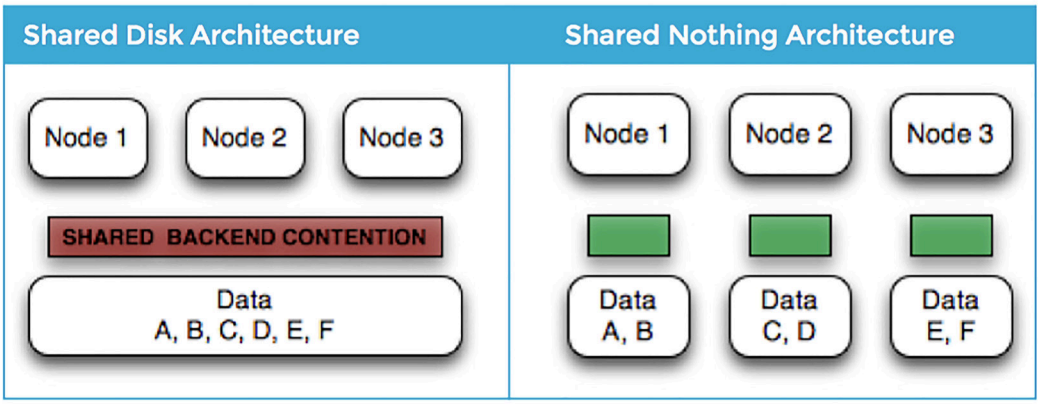
Core ClustrixDB Components to Achieve Near-Linear Scale-Out

- **Shared-nothing architecture** eliminates potential bottlenecks. Contrast this with shared-disk or shared-cache architectures that bottleneck, don't scale, and are difficult to manage.
- **Rebalancer** ensures optimal data distribution across all nodes.
- **Query Optimizer** executes each query with max parallelism and many simultaneous queries with max concurrency. ClustrixDB query optimization leverages a distributed query planner, compiler, and distributed shared-nothing execution engine as well.
- **Evaluation Model** parallelizes queries, which are distributed to the node(s) with the relevant data. Results are then created as close to the data as possible, then routed back to the requesting node for consolidation and return to the client.
- **Consistency and Concurrency Control** uses a combination of Multi-Version Concurrency Control (MVCC) and 2 Phase Locking (2PL); readers have lock-free snapshot isolation while writers use 2PL to manage conflict.



Shared-Nothing Data Distribution

Distributed database systems fall into two major categories of data storage architectures:



Shared disk approaches have several inherent architectural limitations coordinating access to a single central resource. As the number of cluster nodes increases so does coordination overhead. While some workloads can scale well with shared disk, such as small working sets dominated by heavy reads; however most workloads do not, especially heavy write workloads.

ClustrixDB uses the shared-nothing architecture, which has become the architecture of choice for many highly scalable distributed systems (theoretically unlimited) including object storage systems. Shared-nothing designs are highly efficient and solve two fundamental RDBMS problems:

1. Splitting a large data set across any number of individual nodes;
2. Creating an evaluation model that takes advantage of that distributed data environment.

Independent index distribution is the method utilized by ClustrixDB, which enables each index to have its own distribution. Independent index distribution supports a flexible and broad range of distributed query evaluation plans.

ClustrixDB Distribution Concepts	
Representation	Each table contains one or more indexes that are *representations* of the table. Each representation has its own *distribution key* (a.k.a. a partition key), meaning that ClustrixDB uses multiple independent keys to slice the data in one table.
Slice	ClustrixDB breaks each representation into a collection of logical *slices* using consistent hashing. Consistent hashing permits ClustrixDB to split individual slices without having to rehash the entire representation.
Replica	ClustrixDB maintains multiple copies of data for fault tolerance and availability. There are at least two physical *replicas* of each logical slice, stored on separate nodes. The number of *replicas* is selectable to increase DBMS resilience to drive or node failures.

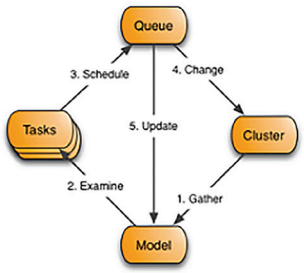
ClustrixDB hashes each distribution key to a 64-bit number space divided into ranges with a specific slice owning each range. The Rebalancer then assigns slices to available nodes in the cluster for data capacity and data access balance.

Rebalancer

The ClustrixDB Rebalancer is an automated system that maintains the healthy distribution of data in the cluster. The Rebalancer is an online process, which effects change to the cluster with minimal interruption to user operations, thus relieving the cluster administrator from the burden of manually manipulating data placement.

There are several components that make up the ClustrixDB Rebalancer:

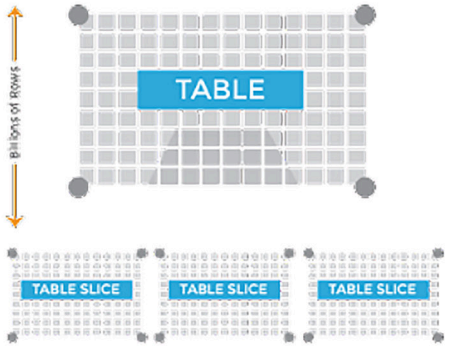
- Information is gathered to build a Model of the cluster's state.
- Tasks examine the Model and decide if action is necessary.
- Tasks post operations to the Queue that will schedule them.



Clustrix has been awarded two patents for distributing and slicing data.

ClustrixDB system user tables are vertically partitioned in representations. Representations are horizontally partitioned into slices. When a new representation is created, the system tries to determine distribution and placement of the data such that:

- The representation has an appropriate number of slices.
- The representation has an appropriate distribution key, to fairly balance rows across its slices, but still allow fast queries to specific slices.
- Slices are well distributed around the cluster on storage devices that are not overfull.
- Slices are not placed on server(s) that are being flexed-down.
- Reads from each representation are balanced across the representation's nodes.



Representations can and do lose these properties over time as their data changes or cluster membership changes. The following are situations that the Rebalancer is designed to remedy:

- **Flex up:** When a node is to be added to the cluster, the administrator will install ClustrixDB on that node, and add that IP to the cluster. The Rebalancer will begin redistributing replicas to that new node.
- **Flex down:** When a node is to be removed from the cluster, the administrator designates it as soft-failed. The Rebalancer will begin moving replicas from this node to other nodes in the cluster.
- **Under protection:** When a slice has fewer replicas than desired, the Rebalancer will create a new copy of the slice on a different node. This can happen due to a storage error, or if a node leaves the cluster quorum because of a local hardware failure or network partition.
- **Load imbalance:** If the slices of a representation are not well distributed across the cluster, the Rebalancer will take steps to move them to more optimal locations. The Rebalancer evaluates as follows: each slice of a representation is assumed to exert load proportional to its share of the representation's key-space. The representation is well distributed when the difference between the "most loaded" and "least loaded" nodes is minimal.
- **Disk too full:** If any storage device in the cluster is holding more than its share of table data, the Rebalancer will take steps to move replicas from that device to an underutilized device. Before moving any replicas, the Rebalancer computes the load imbalance of the cluster's storage devices. If this imbalance is below a configurable threshold, the Rebalancer will leave things alone, avoiding unnecessary and/or non-performant replica moves.
- **Slice is too big:** Representations are partitioned into slices, each of which is assigned a portion of the representation's rows. If a slice becomes large, the Rebalancer will split the slice into several new slices and distribute the original slice's rows among them.
- **Poor distribution:** The rows of a representation are distributed among its slices by the hash value of its distribution key. The distribution key is some prefix of the representation's key columns. Using fewer columns for the distribution key can make certain queries faster because a query that would normally need to broadcast to all slices can instead be targeted to a specific slice. However, if the ratio of rows to unique values in a representation's distribution key is high, some slices can be much larger than other slices for a representation. The Rebalancer corrects this imbalance.
- **Read imbalance:** Reads from a slice are always directed to a select replica, known as the ranking replica. Even if the replicas of a representation are well distributed, the distribution of ranking replicas can be suboptimal. The Rebalancer then adjusts replica rankings so that reads will be well distributed around the cluster.

Query Optimizer

ClustrixDB's Query Optimizer executes a single query with maximum parallelism and many simultaneous queries with maximum concurrency. This is achieved via a distributed query planner and compiler and a distributed shared-nothing execution engine.

Because SQL is a declarative language that describes **what** is to be computed but not **how**, the Query Optimizer determines how to do this computation. That **how** is critical to the performance of the entire system. Take the example of SQL attempting to join three tables and compute an aggregate operation. The Query Optimizer must answer the following questions:

In what order should the tables be joined? This can be the difference between the query executing in 1ms or 10 minutes. Take the example of a predicate on one of the tables causing it to return no rows. Starting the read from that table is likely to be optimal and fast.

Which indexes should be used? Not using a proper index on a join constraint could be catastrophic, causing broadcast messages and full reads of the second table for each row of the first.

Should the sort/aggregate be non-blocking? Should the sort/aggregate be done in stages, i.e. first on separate nodes and then retire-aggregate/re-sort later?

These permutations create a set of query plans known as Search Space. The Query Optimizer explores the Search Space determining which plan uses the least amount of database resources. The most common method is to assign costs to each plan, and choose the least expensive.

The ClustrixDB Query Optimizer is modeled on the Cascades Query optimization framework:

- Cost-driven;
- Extensible via a rule-based mechanism;
- Top-down approach;
- General separation of logical vs. physical operators and properties;
- Branch-and-bound pruning.

The Cascades framework is production proven, satisfying many commercial DBMS systems, notably Tandem's Nonstop SQL and Microsoft's SQL Server.

Modern query optimizers are often split into the Model and Search Engine. The Model lists the equivalence transformations (rules) used by the search engine to expand the search space. The Search Engine defines the interfaces between the search engine and the model. It expands the search space, searches for the optimal plan, and is implemented by the tasks stack waiting to be computed.

ClustrixDB Query Optimizer Core Components

Logical model describes **what** is to be computed and Physical model describes **how** it's to be computed.

An **expression** consists of:

- An Operator (required);
- Arguments (possibly none);
- Inputs (possibly none).

Arguments describe particular characteristics of the operator. There are both logical and physical operators. Every logical operator maps to one or more physical operators.

Physical properties are related to intermediate results, or sub-plans. They describe things like how the data is ordered and how the data is partitioned. It is important to note that logical or physical expressions and groups (see below) do not have physical properties. However, every physical expression has two descriptions related to physical properties:

1. What Physical properties can and can't be provided to the parent?
2. What Physical properties are required for the input?

Groups correspond to intermediate tables, or equivalent subplans, of the query. Groups are logical and contain the following:

1. All the logically equivalent expressions that describe that intermediate table
2. All the physical implementations of those logical expressions
3. Winners: A physical expression that had the best cost given a set of physical properties
4. Logical properties: Which columns and statistics about some columns it is required to produce
5. Groups are the fundamental data structure in the ClustrixDB Query Optimizer. The inputs to operators are always groups (indicated by group #'s), and every expression corresponds to some group

Memo: The ClustrixDB Query Optimizer keeps track of the intermediate tables that can be used in computing the final result table while optimizing. Each of these corresponds to a group and the set of all groups for a plan defines the memo. The memo is designed to represent all logical query trees and physical plans in the search space for a given initial query. The memo is a set of groups with one group designated as the final (or top) group. This is the group corresponding to the table of results from the evaluation of the initial query. The optimizer has no explicit representation of all possible query trees. Instead, this memo represents a compact version of all possible query trees.

Rules: The model's rule set defines the logical and physical search space of the optimizer. The memo is expanded to encompass the full logical and physical search space through the application of rules. The application of rules is a multi-step process of finding a binding in the memo, evaluating a rule condition (if the rule has one) and (if the condition is satisfied) firing the rule, which puts new expressions in the memo. When the optimizer is done applying rules, the memo structure will have been expanded to where it conceptually represents the entire search space.

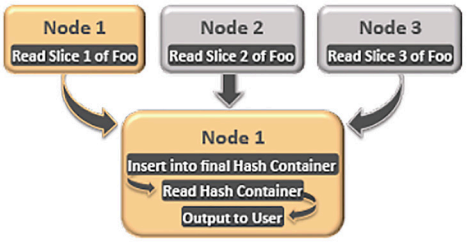
Tasks: There are tasks waiting on a stack to be executed at any point in time during optimization. Each task frequently pushes additional tasks onto the stack to achieve its goal. The optimizer is done computing once the stack is empty. It begins by taking an input tree and constructing the corresponding initial groups and expressions. It then starts off the search engine by pushing the task Optimize_group (top group). This causes a chain of events that explores the entire search space, finds the cheapest winners for each group, and finally chooses the cheapest winner in the top group to be its output plan.

Cost model: The optimizer costs plans use a combination of I/O, CPU usage, and latency. Because ClustrixDB is distributed, total CPU usage and latency are not proportional. Every operator describes a function to compute its costs given its inputs. The optimizer chooses the optimal plan for a query by finding the plan with the cheapest cost. Cost is strongly dependent on how many rows the optimizer thinks are going to be flowing through the system. The job of the row estimation subsystem is to take statistical information from our Probability Distributions ("PDs") and compute an estimated number of rows that will come out of a given expression.

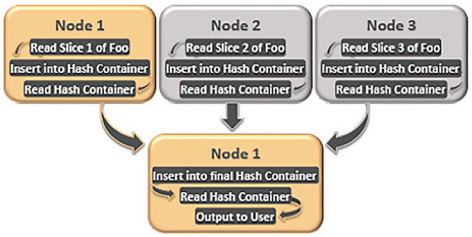
Distributed considerations: One of the special things about the optimizer is its ability to reason about doing distributed operations. There are two ways to compute an aggregate non-distributed and distributed. In the distributed approach:

1. Read table Foo which likely has slices on multiple nodes
2. Forward those rows to one node
3. Insert those rows into a hash container, computing the aggregate operation if necessary
4. Read the container
5. Output to the user

When the aggregate is distributed instead:



1. Compute the sub-plan (under the aggregate), which likely has result rows on multiple nodes
2. Locally insert those rows into a local hash container, computing the aggregate operation if necessary
3. Read the hash container on each node and forward to a single node
4. If necessary insert all those rows into a new final hash container, computing the aggregate operation
5. Read that hash container
6. Output rows to the user



The optimizer must determine which one is better and when. The gains from distributing the aggregate actually come from potentially sending a lot less data across the network (between nodes). The additional overhead of extra inserts and containers is less than the network latency gains when the reduction factor of the aggregate operation is large.

The optimizer is able to calculate this with the cost model and determine the better approach for any query, which is part of the Distributed Aggregates feature.

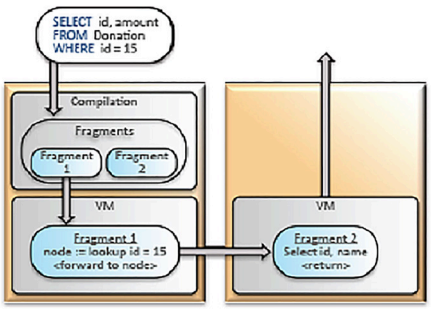
Evaluation Model

The next key ClustrixDB functionality is its ability to 'send the query to the data.' This is one of the fundamental principles of how ClustrixDB can scale near linearly as more nodes are added. Other RDBMS systems routinely move large amounts of data to the node that is processing the query, then eliminate all the data that don't fit the query (often lots of data), whereas ClustrixDB appreciably reduces network traffic performance issues by only moving qualified data to the requesting node. Processors on multiple nodes can additionally be brought to bear on the data selection process. The system produces results more quickly by selecting data on multiple nodes in parallel rather than selecting all data from a single node that must first gather the required data from the other nodes in the system.

ClustrixDB uses parallel query evaluation for simple queries and Massively Parallel Processing (MPP) for analytic queries (akin to columnar stores). The Fair Scheduler additionally ensures that OLTP queries are prioritized ahead of OLAP queries. Data is read from the ranking replica assigned by the Rebalancer. This can either reside on the same node or require at most a single hop. The number of hops that one query requires (0 or 1) doesn't change as data set size and the number of nodes increase, enabling linear scalability of both writes and queries.

Query Compilation to Fragments

Queries are broken down during compilation into fragments that are analogous to a collection of functions. For example, a simple query can be compiled into two functions: The first function looks up where the value resides and the second function reads the correct value from the container on that node and slice and returns to the user (the details of concurrency, etc. have been left out for clarity).



Scaling Joins

Joins require more data movement by their nature. ClustrixDB is able to achieve minimal data movement even in complex joins because:

- Each representation (table or index) has its own distribution map, allowing direct look-ups for which node/slice to go to next, removing broadcasts.
- There is not a central node orchestrating data motion. Data moves directly to the next node it needs to go to. This reduces hops to the minimum possible given the data distribution.

Concurrency Control

ClustrixDB uses a combination of Multi-Version Concurrency Control (MVCC) and 2 Phase Locking (2PL) to support mixed read-write workloads. Readers have 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.

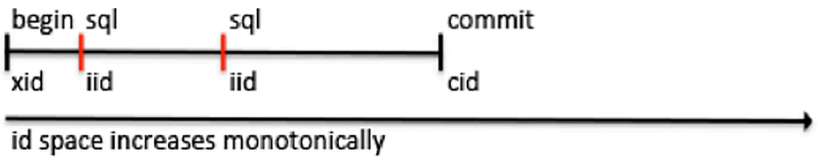
Multi-Version Concurrency Control

ClustrixDB implements a distributed MVCC scheme to ensure that readers are lockless ensuring readers and writers never interfere with each other. ClustrixDB maintains a version history of each row as writers modify rows within the system. 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

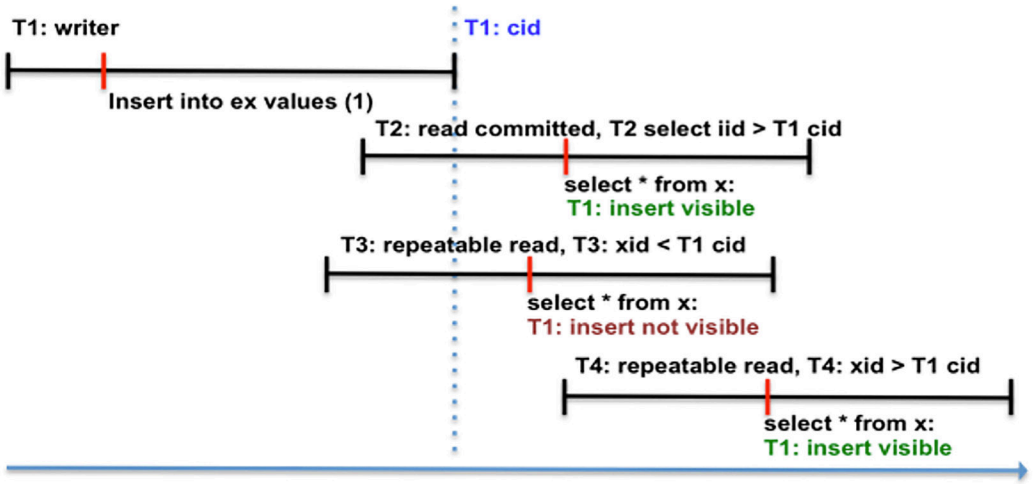


xid	Transaction start id. Marks the logical beginning of the transaction.
iid	Invocation id. Marks the beginning of a statement within a transaction.
cid	Commit id. Marks the id at which transaction changes are visible to other transactions. The following table describes MVCC visibility rules in different 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.



This example illustrates how transaction visibility works at different isolation levels

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). A purely MVCC system would roll back one or both of the conflicting transactions and restart the operation when that occurs. ClustrixDB does not require predetermined transactions (e.g. all logic within a stored procedure). Therefore errors can potentially bubble up to the application. It's also possible to create live-lock scenarios where transactions cannot make progress because of constant conflict.

ClustrixDB overcomes these issues through 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. Each node maintains a portion of the lock domain within the cluster. No single node holds all cluster lock information.

Row Level & Table Level Locking

ClustrixDB implements row level locks for transactions that touch a few rows at a time (a run time configurable variable). The Query Optimizer will promote row level locks to a table lock for statements that affect a signification portion of a table.

Consistency

Many distributed databases have embraced eventual consistency over strong consistency. They do it because eventual consistency reduces system scalability complexity. The significant downside is that anomalies that arise put the burden of fixing them on the application. Eventual consistency comes at the very high cost of increased manually labor-intensive programming model complexity.

ClustrixDB delivers a consistency model that scales using a combination of intelligent data distribution, multi-version concurrency control (MVCC), and Paxos. This enables ClustrixDB to scale writes, scale reads (queries) in the presence of write workloads, and provide strong ACID semantics. ClustrixDB delivers scalable consistency through the use of:

- Synchronous replica on within the cluster. All participating nodes must acknowledge writes before the write can complete. Writes performed in parallel.
- Paxos protocols for distributed transaction resolution.
- Support for Read Committed, Repeatable Read (Snapshot) isolation levels. Limited support for Serializable.
- MVCC allows for lockless reads; writes will not block reads.

For a detailed in-depth explanation of how ClustrixDB scales reads and writes, refer to the ClustrixDB concurrency model architecture documentation.

How ClustrixDB Provides Fault Tolerance, HA, & No SPOF

ClustrixDB fault tolerance comes from maintaining multiple copies of data across the cluster. The degree of fault tolerance depends on the specified number of replicas (minimum of two). Each replica is placed on a different node protecting the data from multiple concurrent hardware faults. For more details on how ClustrixDB handles fault tolerance please read the ClustrixDB data distribution model documentation.

ClustrixDB offers high availability even in the face of failure. In order to provide that full availability, ClustrixDB requires that:

- A majority of nodes are able to form a cluster (i.e. quorum requirement).
- The available nodes hold at least one replica for each set of data.

Specifically, with all the database's relations at the default of replicas=2, this means that a 3-node cluster could lose a single node, and still be fully available. The Rebalancer will automatically engage and depending on free disk space the cluster will again become fully redundant at a reduced performance. When adding an additional node to that 2-node cluster, the new node's processors will immediately be utilized for incoming queries, while the Rebalancer begins redistributing replicas evenly across 3 nodes.

A 3-node cluster with replicas=3 can lose 2 nodes without losing data, but now it has lost redundancy. ClustrixDB will alert the user that redundancy is lost, and recommend adding at least 1 additional node.

Note that no matter what the table definitions specify, the Rebalancer will rectify that with the amount of nodes available. So in the case of a single node in the fault-tolerance scenario above, there will only be a single replica of each table data-slice. When an additional node is added, the Rebalancer will create a second replica, and update the data-map (each node's data-map contains the location of every replica in the system, and identifies which are the ranking replicas). And then when a 3rd node is added to the cluster, the Rebalancer will create that 3rd replica, regaining full multi-node failure tolerance. And it does this without changing the original table definition.

In Summary

ClustrixDB is able to natively scale-out transactions in a clustered database, providing high availability and fault tolerance while being wire-compatible with MySQL. It accomplishes this by leveraging a shared-nothing architecture, automatic data slicing and distribution by the Rebalancer, Query Optimization, and an Evaluation Model, which distributes pre-compiled query fragments to the node on which the data resides. In addition, MVCC and 2 Phase Locking (2PL) are leveraged. All of this was written from the ground-up, and does not leverage any MySQL code.

When compared to other common workarounds to scale traditional RDBMS', there is no comparison.

ClustrixDB ACID Guarantees

ClustrixDB Atomicity

ClustrixDB uses a combination of two-phase locking and multi-version concurrency control (MVCC) to ensure atomicity. Whereas this can increase latency if every node participated in every transaction, ClustrixDB avoids that problem through the use of the Paxos consistency protocol. The Paxos consistency protocol compels transaction participants to include the originating node, three logging nodes, and the nodes where data is stored. A single node may serve multiple functions in a given transaction, ensuring that simple point selects and updates have a constant overhead. OLAP transactions compute partial aggregates on the node where the data is located and therefore similarly don't suffer from high overhead.

ClustrixDB Consistency

ClustrixDB's shared-nothing architecture is immediately consistent, which is significantly different from NoSQL shared-nothing architectures that are only "eventually" consistent. Eventual consistency can produce significant

errors that can invalidate or corrupt a relational database. ClustrixDB delivers immediate consistency by ensuring relational constraints, such as foreign keys, are enforced properly by implementing those foreign keys as triggers and evaluating at commit time. As a result, clients connecting to any node in the cluster to issue queries will see the same data across all nodes.

ClustrixDB Isolation

ClustrixDB produces MVCC isolation at the container level. The ClustrixDB atomicity guarantees that all applicable replicas receive a particular write before it reports the transaction committed. This makes ClustrixDB isolation equivalent to non-clustered MySQL isolation. And unlike other systems like Galera Cluster, ClustrixDB snapshot isolation does use 'first-committer wins.'

ClustrixDB Durability

ClustrixDB provides ACID durability via normal write ahead logging (WAL) as well as replication of relational data within the cluster. Durable consistency is ensured by hash distributing relational data across nodes on a per-index basis as 'slices.' Each slice has a minimum of two replicas (more are optional) spread throughout the cluster as protection against multiple concurrent nodes or drive failures.