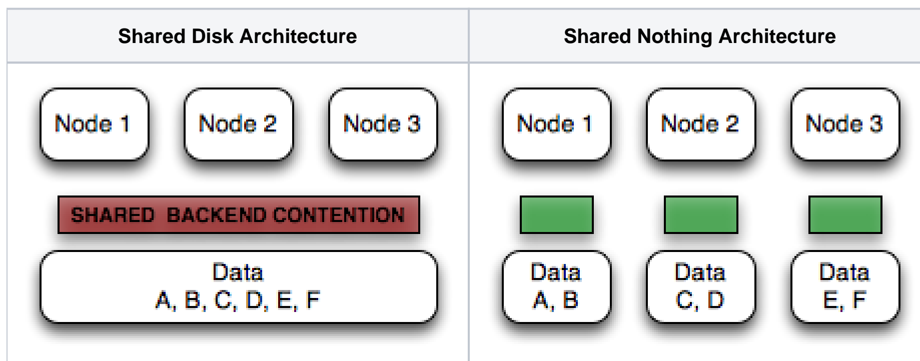
Data Distribution

- Introduction
  - Shared Disk vs. Shared Nothing
  - Shared Nothing Challenges
- Shared Nothing Distribution Strategies
  - ClustrixDB Basics
    - Distribution Concepts Overview
    - Concepts Example
    - Representation
    - Slice
    - Replica
- Consistent Hashing
  - Slicing
  - Re-Slicing For Growth
- Single Key vs. Independent Index Distribution
  - Single Key Approach
  - Independent Index Key Approach
- Cache Efficiency

Introduction

Shared Disk vs. Shared Nothing

Distributed database systems fall into two major categories of data storage architectures: (1) shared disk and (2) shared nothing.



Shared disk approaches suffer from several architectural limitations inherent in coordinating access to a single central resource. In such systems, as the number of nodes in the cluster increases, so does the coordination overhead. While some workloads can scale well with shared disk (e.g. small working sets dominated by heavy reads), most workloads tend to scale very poorly -- especially workloads with significant write load.

ClustrixDB uses the shared nothing approach because it's the only known approach that allows for large-scale distributed systems.

Shared Nothing Challenges

In order to build a scalable shared-nothing database system, one must solve two fundamental problems:

1. Split a large data set across a number of individual nodes.
2. Create an evaluation model that can take advantage of the distributed data environment.

This document explains how ClustrixDB distributes data sets across a large number of independent nodes, as well as provides reasoning behind some of our architectural decisions.

Shared Nothing Distribution Strategies

Within shared nothing architectures, most databases fall into the following categories:

1. **Table-level distribution**. The most basic approach, where an entire table is assigned to a node. The database does not split the table. Such systems cannot handle very large tables.
2. **Single-key-per-table distribution (a.k.a index colocation, or single-key sharding)**. The most common approach. The preferred method for most distributed databases (e.g. MySQL Cluster, MongoDB, etc.). In this approach, the table is split into multiple chunks using a single key (user id, for example). All indexes associated with the chunk are maintained (co-located) with the primary key.
3. **Independent index distribution**. The strategy used by ClustrixDB. In this approach, each index has its own distribution. Required to support a broad range of distributed query evaluation plans.

ClustrixDB Basics

ClustrixDB has a fine-grained approach to data distribution. The following table summarizes the basic concepts and terminology used by our system. Notice that unlike many other systems, ClustrixDB uses a per-index distribution strategy.

Distribution Concepts Overview

ClustrixDB Distribution Concepts	
Representation	Each table contains one or more indexes. Internally, ClustrixDB refers to these indexes as *representations* of the table. Each representation has its own *distribution key* (a.k.a. a partition key or a shard key), meaning that ClustrixDB uses multiple independent keys to slice the data in one table. This is in contrast to most other distributed database systems, which use a single key to slice the data in one table. Each table must have a primary key. If the user does not define a primary key, ClustrixDB will automatically create a hidden primary key. The *base representation* contains all of the columns within the table, ordered by the primary key. Non-base representations contain a subset of the columns within the table.
Slice	ClustrixDB breaks each representation into a collection of logical *slices* using consistent hashing. By using consistent hashing, ClustrixDB can 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. ClustrixDB supports configuring the number of replicas per representation. For example, a user may require three replicas for the base representation of a table, and only two replicas for the other representations of that table.

Concepts Example

Consider the following example:

```
sql> CREATE TABLE example (
        id      bigint        primary key,
        col1        integer,
        col2        integer,
        col3    varchar(64),
        key k1  (col2),
        key k2  (col3, col1)
);
```

We populate our table with the following data:

Table: example			
id	**col1**	**col2**	**col3**
1	16	36	january
2	17	35	february
3	18	34	march
4	19	33	april
5	20	32	may

Representation

ClustrixDB will organize the above schema into three *representations*. One for the main table (the *base representation*, organized by the primary key), followed by two more representations, each organized by the index keys.

The yellow coloring in the diagrams below illustrates the ordering key for each representation. Note that the representations for the secondary indexes include the primary key columns.

Table: example		
base representation	**k1 representation**	**k2 representation**

primary key				index (col2)		index (col3, col1)		
id	**col1**	**col2**	**col3**	**col2**	**id**	**col3**	**col1**	**id**
1	16	36	january	32	5	april	19	4
2	17	35	february	33	4	february	17	2
3	18	34	march	34	3	january	16	1
4	19	33	april	35	2	march	18	3
5	20	32	may	36	1	may	20	5

Slice

ClustrixDB will then split each representation into one or more logical *slices*. When slicing ClustrixDB uses the following rules:

- We apply a consistent hashing algorithm on the representation's key.
- We distribute each representation independently. Refer to single key vs. independent distribution below for an in-depth examination of the reasoning behind this design.
- The number of slices can vary between representations of the same table.
- We split slices based on size.
- Users may configure the initial slice count of each representation. By default, each representation starts with one slice per node.

base representation slices											
slice 1				slice 2				slice 3			
id	**col1**	**col2**	**col3**	**id**	**col1**	**col2**	**col3**	**id**	**col1**	**col2**	**col3**
2	17	35	february	1	16	36	january	3	18	34	march
4	19	33	april	5	20	32	may				

k1 representation			
slice 1		slice 2	
col2	**id**	**col2**	**id**
32	5	33	4
34	3	36	1
35	2		

k2 representation											
slice 1			slice 2			slice 3			slice 4		
col3	**col2**	**id**	**col3**	**col2**	**id**	**col3**	**col2**	**id**	**col3**	**col2**	**id**
april	19	4	february	17	2	january	16	1	may	20	5
						march	18	3			

Replica

To ensure fault tolerance and availability ClustrixDB contains multiple copies of data. ClustrixDB uses the following rules to place *replicas* (copies of slices) within the cluster:

- Each logical slice is implemented by two or more physical replicas. The default protection factor is configurable at a per-representation level.
- Replica placement is based on balance for size, reads, and writes.
- No two replicas for the same slice can exist on the same node.
- ClustrixDB can make new replicas online, without suspending or blocking writes to the slice.

Sample data distribution within a 4 node cluster			
node 1	**node 2**	**node 3**	**node 4**

k2 slice 1 replica A

col3	col2	id
april	19	4

k2 slice 2 replica B

col3	col2	id
february	17	2

base rep slice 3 replica A

id	col1	col2	col3
3	18	34	march

base rep slice 2 replica B

id	col1	col2	col3
1	16	36	january
5	20	32	may

k2 slice 3 replica A

col3	col2	id
january	16	1
march	18	3

k2 slice 1 replica B

col3	col2	id
april	19	4

base rep slice 1 replica A

id	col1	col2	col3
2	17	35	february
4	19	33	april

k2 slice 2 replica A

col3	col2	id
february	17	2

k2 slice 4 replica B

col3	col2	id
may	20	5

base rep slice 2 replica A

id	col1	col2	col3
1	16	36	january
5	20	32	may

base rep slice 3 replica B

id	col1	col2	col3
3	18	34	march

k2 slice 4 replica A

col3	col2	id
may	20	5

k2 slice 3 replica B

col3	col2	id
january	16	1
march	18	3

base rep slice 1 replica B

id	col1	col2	col3
2	17	35	february
4	19	33	april

Consistent Hashing

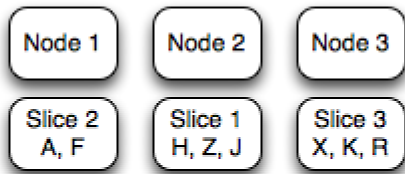
ClustrixDB uses consistent hashing for data distribution. Consistent hashing allows ClustrixDB to dynamically redistribute data without having to rehash the entire data set.

Slicing

ClustrixDB hashes each distribution key to a 64-bit number. We then divide the space into ranges. Each range is then owned by a specific slice. The table below illustrates how consistent hashing assigns specific keys to specific slices.

Slice	Hash Range	Key Values
1	min-100	H, Z, J
2	101-200	A, F
3	201-max	X, K, R

ClustrixDB then assigns slices to available nodes in the Cluster for data capacity and data access balance.



Re-Slicing For Growth

As the dataset grows, ClustrixDB will automatically and incrementally re-slice the dataset one or more slices at a time. We currently base our re-slicing thresholds on data set size. If a slice exceeds a maximum size, the system will automatically break it up into two or more smaller slices.

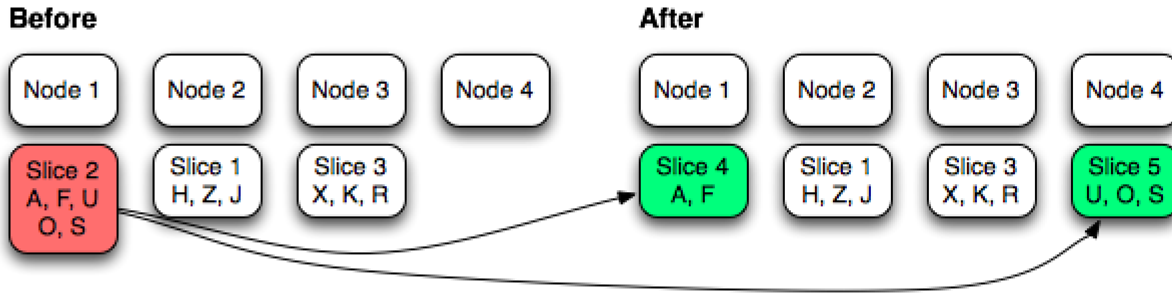
For example, imagine that one of our slices grew beyond the preset threshold:

Slice	Hash Range	Key Values	Size
1	min-100	H, Z, J	768MB
2	101-200	A, F, U, O, S	1354MB (too large)
3	201-max	X, K, R, Y	800MB

Our rebalancer process will automatically detect the above condition and schedule a slice-split operation. The system will break up the hash range into two new slices:

Slice	Hash Range	Key Values	Size
1	min-100	H, Z, J	768MB
~~2~~	~~101-200~~	~~A, F, U, O, S~~	~~1354MB (too large)~~
4	101-150	A, F	670MB
5	151-200	U, O, S	684MB
3	201-max	X, K, R, Y	800MB

Note that system does not have to modify slices 1 and 3. Our technique allows for very large data reorganizations to proceed in small chunks.



Single Key vs. Independent Index Distribution

It's easy to see why table-level distribution provides very limited scalability. Imagine a schema dominated by one or two very large tables (billions of rows). Adding nodes to the system does not help in such cases since a single node must be able to accommodate the entire table.

Why does ClustrixDB use independent index distribution rather than a single-key approach? The answer is two-fold:

1. Independent index distribution allows for a much broader range of distributed query plans that scale with cluster node count.
2. Independent index distribution requires strict support within the system to guarantee that indexes stay consistent with each other and the main table. Many systems do not provide the strict guarantees required to support index consistency.

Let's examine a specific use case to compare and contrast the two approaches. Imagine a bulletin board application where different topics are grouped by threads, and users are able to post into different topics. Our bulletin board service has become popular, and we now have billions of thread posts, hundreds of thousands of threads, and millions of users.

Let's also assume that the primary workload for our bulletin board consists of the following two access patterns:

1. Retrieve all posts for a particular thread in post id order.
2. For a specific user, retrieve the last 10 posts by that user.

We could imagine a single large table that contains all of the posts in our application with the following simplified schema:

```
-- Example schema for the posts table.
sql> CREATE TABLE thread_posts (
        post_id         bigint,
        thread_id        bigint,
        user_id         bigint,
        posted_on        timestamp,
        contents        text,
        primary key     (thread_id, post_id),
        key             (user_id, posted_on)
);

-- Use case 1: Retrieve all posts for a particular thread in post id order.
-- desired access path: primary key (thread_id, post_id)
sql> SELECT *
     FROM  thread_posts
     WHERE thread_id = 314
     ORDER BY post_id;

-- Use case 2: For a specific user, retrieve the last 10 posts by that user.
-- desired access path: key (user_id, posted_on)
sql> SELECT *
     FROM thread_posts
     WHERE user_id = 546
     ORDER BY posted_on desc
     LIMIT 10;
```

Single Key Approach

With the single key approach, we are faced with a dilemma: Which key do we choose to distribute the posts table? As you can see with the table below, we cannot choose a single key that will result in good scalability across both use cases.

Distribution Key	Use case 1: posts in a thread	Use case 2: top 10 posts by user
thread_id	Queries that include the thread_id will perform well. Requests for a **specific thread get routed to a single node** within the cluster. When the number of threads and posts increases, we simply add more nodes to the cluster to add capacity.	Queries that do not include the thread_id, like the query for last 10 posts by a specific user, must evaluate on all nodes that contain the t hread_posts table. In other words, the system must **broadcast** the query request because the relevant post can reside on any node.
user_id	Queries that do not include the user_id result in a broadcast. As with the thread_id key sample (use case 2), we lose system scalability when we have to broadcast.	Queries that include a user_id get routed to a single node. Each node will contain an ordered set of posts for a user. The system can scale by avoiding broadcasts.

One possibility with such a system could be to maintain a separate table that includes the user_id and posted_on columns. We can then have the application manually maintain this index table.

However, that means that the application must now issue multiple writes, and accept responsibility for data consistency between the two tables. And imagine if we need to add more indexes? The approach simply doesn't scale. One of the advantages of a database is automatic index management.

Independent Index Key Approach

ClustrixDB will automatically create independent distributions that satisfy both use cases. The DBA can specify to distribute the base representation (primary key) by thread_id, and the secondary key by user_id. The system will automatically manage both the table and secondary indexes with full ACID guarantees.

For more detailed explanation consult our Evaluation Model section.

Cache Efficiency

Unlike other systems that use master-slave pairs for data fault tolerance, ClustrixDB distributes the data in a more fine-grained manner as explained in the above sections. Our approach allows ClustrixDB to increase cache efficiency by not sending reads to secondary replicas.

Consider the following example. Assume a cluster of 2 nodes and 2 slices A and B, with secondary copies A' and B'.

Read from both copies

Node 1	Node 2
A	B
B'	A'

Read from primary copy only

Node 1	Node 2
A	B
B'	A'

If we allow reads from both primary and secondary replicas, then each node will have to cache contents of both A and B. Assuming 32GB of cache per node, the total effective cache of the system becomes 32GB.	By limiting the reads to primary replica only, we make node 1 responsible for A only, and node 2 responsible for B only. Assuming 32GB cache per node, the total effective cache footprint becomes 64GB, or double of the opposing model.