



# Scaling and the Sharding Deep Freeze

Clustrix has created a scalable database system that grows online through clustering, living across the nodes in the cluster. This paper describes why we are focused on scalability in our product and explores how software architects can start down a sharding path and soon find themselves stuck in a “deep freeze.”

At Clustrix, we've been focused on creating a scalable database system that grows online through clustering. The database itself lives across the nodes in the cluster. This allows the system to not be constrained by the size of any single piece of hardware. This also allows the system to route around faults and be easily managed as a unit.

Most internet infrastructure projects start with the idea of connectedness. This includes concepts like ownership, sharing, notifications and updates, privacy, and community. There is a database layer (usually MySQL), and that database layer is the natural place to hold all of the structured information necessary for the application to represent everything related to the above concepts. We will explore how software architects can start down a sharding path and pretty soon realize they're stuck in a "deep freeze". Everything from then on, from development to deployment takes longer and is a lot more complicated. The effort required to get the most basic features developed once a sharding backend has been introduced continues to go up and up.

Let's simplify things by looking at a basic internet storefront which only cares about users, objects, and orders. This isn't nearly enough functionality for a differentiated site, but it is enough to see where the sharding issues crop up. The storefront

database probably started out with tables which match each of the three concepts: a `users` table, an `objects` table, and an `orders` table. Each user gets an entry in the users table, with an associated user id. Each object that a user wants to sell goes into the objects table, with an object id and an owner\_id which references a row in the users table. Each order has an object\_id, seller\_id, and buyer\_id. Each order represents a transaction of the object from the seller to the buyer. This should look like pretty straightforward e-commerce to most people.

```
CREATE TABLE users (  
    id integer unsigned primary key auto_increment,  
    name varchar(256),  
    account_balance integer unsigned  
);  
  
CREATE TABLE objects (  
    id integer unsigned primary key auto_increment,  
    name varchar(256),  
    owner_id integer unsigned,  
    index owner_id(owner_id)  
);  
  
CREATE TABLE orders (  
    id integer unsigned primary key auto_increment,  
    price integer unsigned,  
    seller_id integer unsigned,  
    buyer_id integer unsigned,  
    object_id integer unsigned,  
    index seller_id(seller_id),  
    index buyer_id(buyer_id),  
    index object_id(object_id)  
);
```

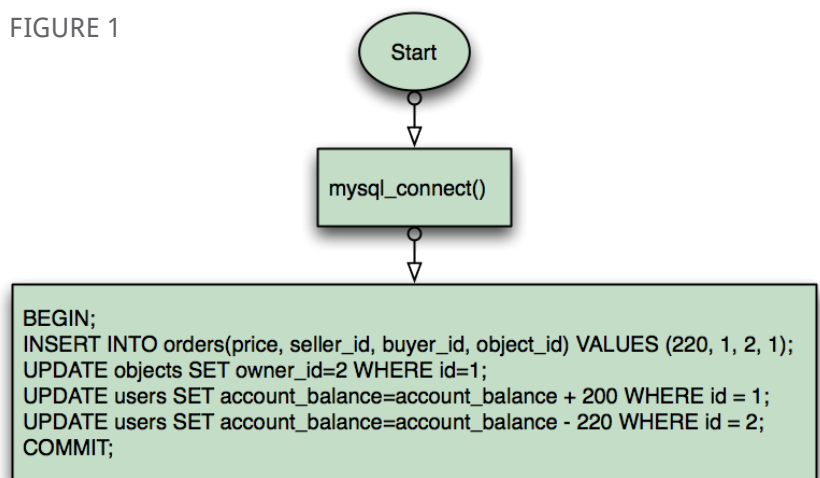
Everything is going along great—our store is growing, users are signing up regularly, business is good. See Figure 1 to the right for the flowchart of our application logic around the database to insert a new order.

The user with id 2 has purchased object id 1 from user id 1 for a price of \$220. The database logic is wrapped up in a single transaction. This transaction either entirely completes or fails. There isn't a state where only part of the data gets written and things wind up out of sync. ACID gives us the atomicity, consistency, isolation, and durability that makes this type of workload happen in a well-defined way.

With the quick uptake and positive customer response, our website is growing and pretty soon we notice things start to slowdown. After a bit of investigation we figure out that it's the database that isn't keeping up with the load. There simply is too much work for a single instance server to handle. This is the point where sharding starts to show up as an option. Instead of having everything in one database, with sharding we break things up into multiple databases. Each individual database instance will have the same set of tables, but will contain a unique set of data. Each individual database is a "shard".

How do we keep track of which users, objects, and orders are in which shard? There are two common ways to do it: a) use some form of hashing on primary keys, or b) a master directory database server. If we choose a) then

FIGURE 1



we have the rehashing problem—what do we do when shards start getting loaded? We have to rehash the sharding complex, most likely taking significant downtime while doing so. With option b) the master directory database server itself will eventually become a bottleneck and we'll be stuck again. Which ever direction we chose, we will need to do custom engineering to design the layout and maintain it.

However, there is an even larger, more constant overhead to sharding no matter which direction you chose for managing the partitioning of the shards. In a sharded infrastructure you no longer have one database to query. There are a bunch of technical data decisions to make which are completely an artifact of the fact that you've sharded. Do the orders live on the shards with the seller or the buyer? Or, are they distributed on their own? How big does a shard get before we want to re-shard? When we re-shard do we break the shard in half? In quarters? Does each shard have the same limit? Are some hotter than others? Do we reshare in the case where some get really hot?

Once we have sharded, this whole thing becomes much more complex. For each of the seller, object, and buyer we have to figure out what shard they are in. This is done either by the algorithmic hash value, or looking up in the master directory database.

We then have to connect to each shard, and do the updates. If things fail on some shards and succeed on others, then we have to back out whichever changes we made and start over again. There's a huge portion of rollback code which needs to be able to unwind things at any stage if any portion of the logical operation fails. If we do everything perfectly, there's at least twice the latency (minimum two independent database updates) and 3x the code (lookup, connection, and undo code).

Let's look at a slightly more complicated example: something that deals with the actual relations in the tables. How about if we want to get the name of every person that "Paul Mikesell" has ever sold to. That would look like Figure 3 to the right if we had a single database system.

Conceptually, using SQL we're asking to match sellers with buyers via the orders table, and constrain things to where the seller is named "Paul Mikesell." This is a pretty straightforward 3 way SQL join. But look at what happens in our hypothesized sharded environment (Figure 4 on the next page).

FIGURE 2

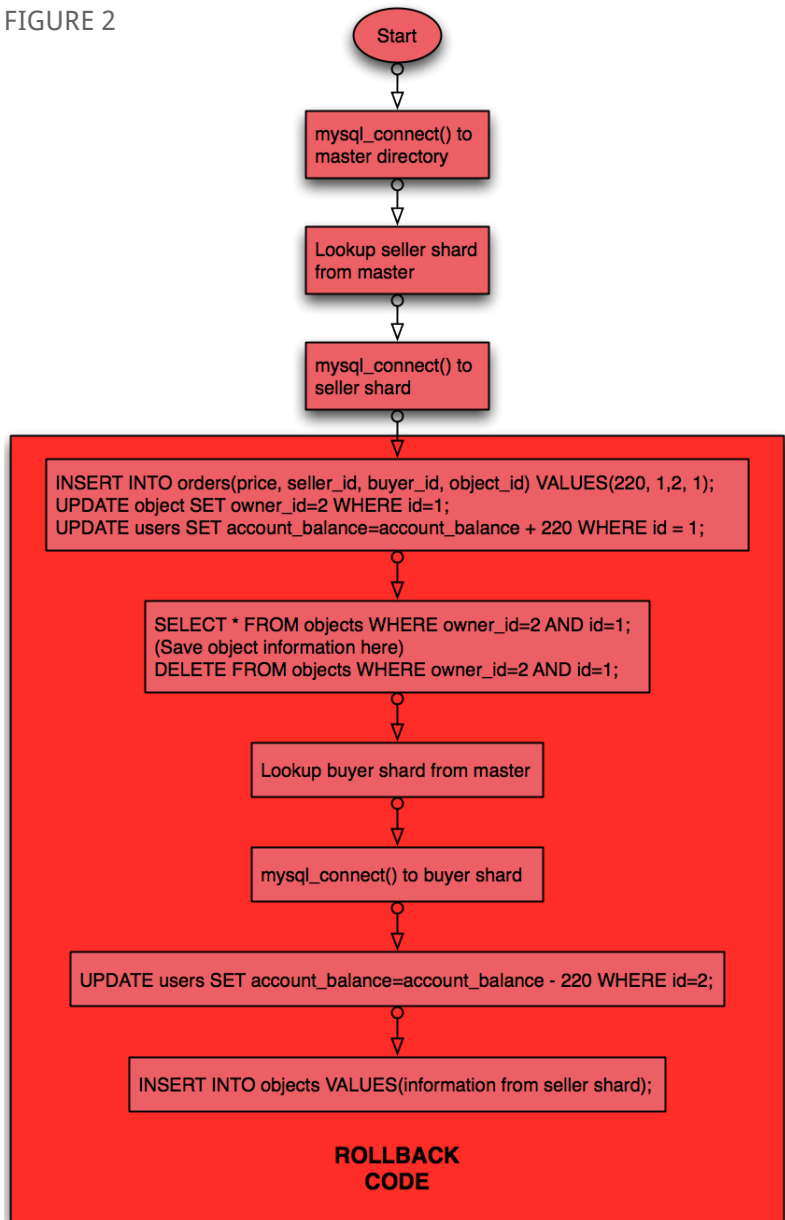


FIGURE 3

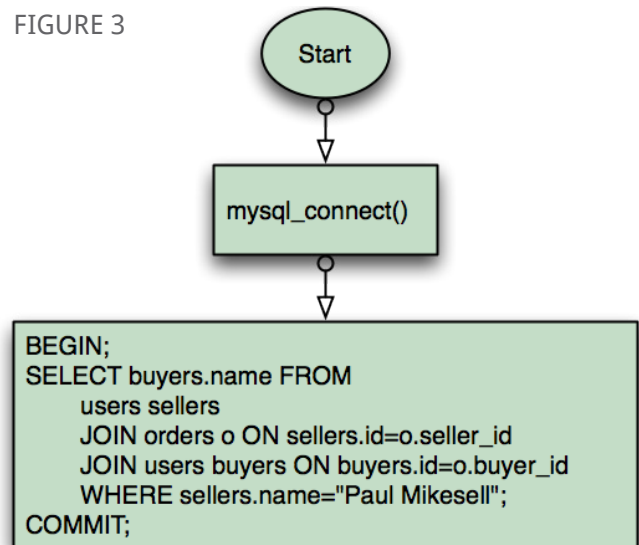
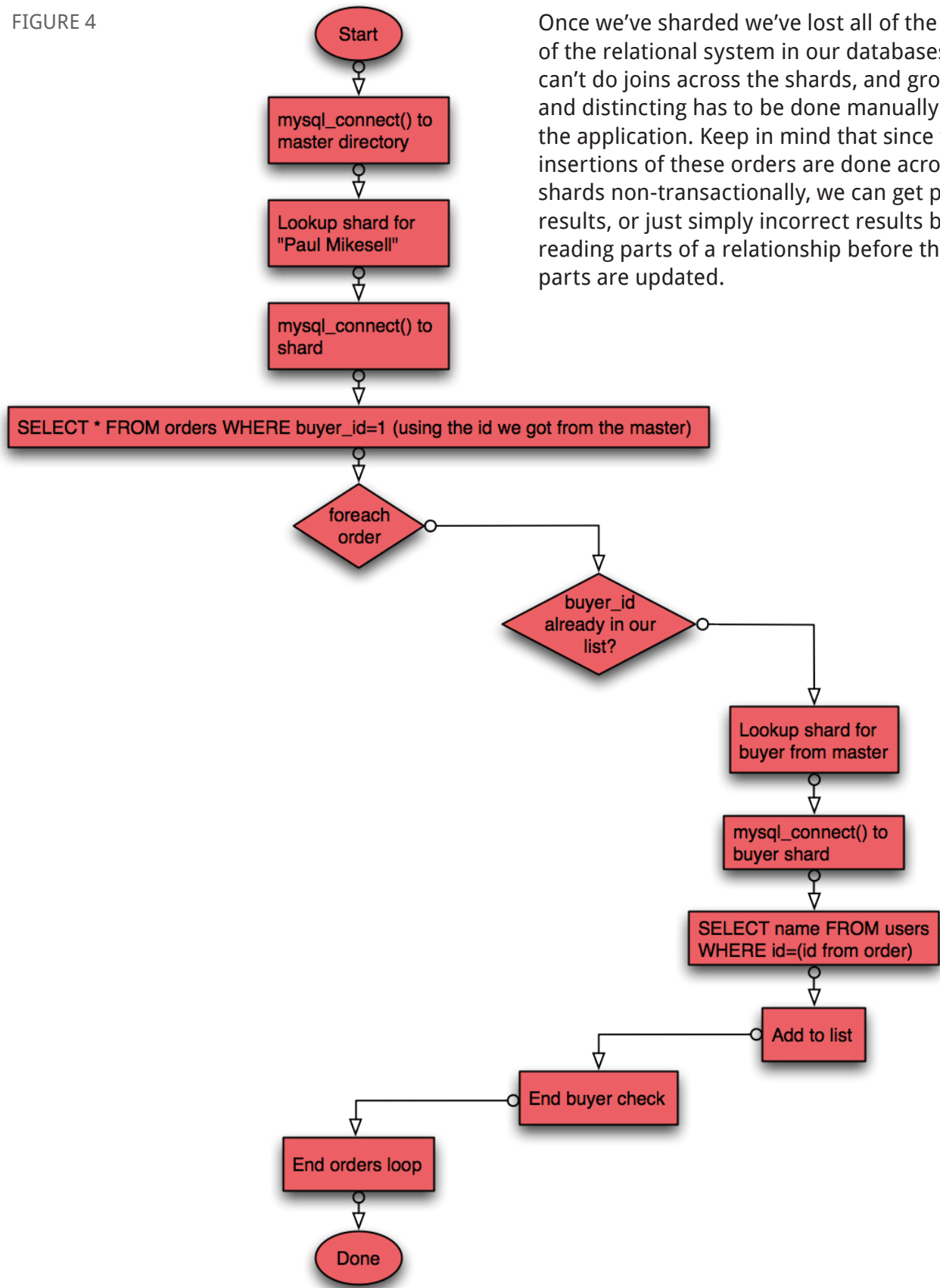


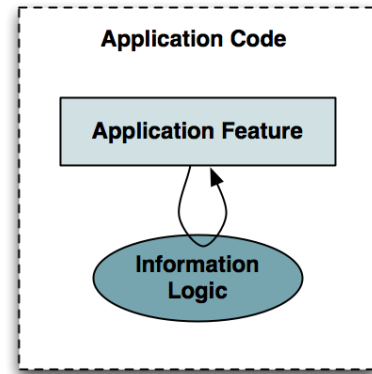
FIGURE 4



Once we've sharded we've lost all of the power of the relational system in our databases. We can't do joins across the shards, and grouping and distincting has to be done manually in the application. Keep in mind that since the insertions of these orders are done across shards non-transactionally, we can get partial results, or just simply incorrect results by reading parts of a relationship before the other parts are updated.

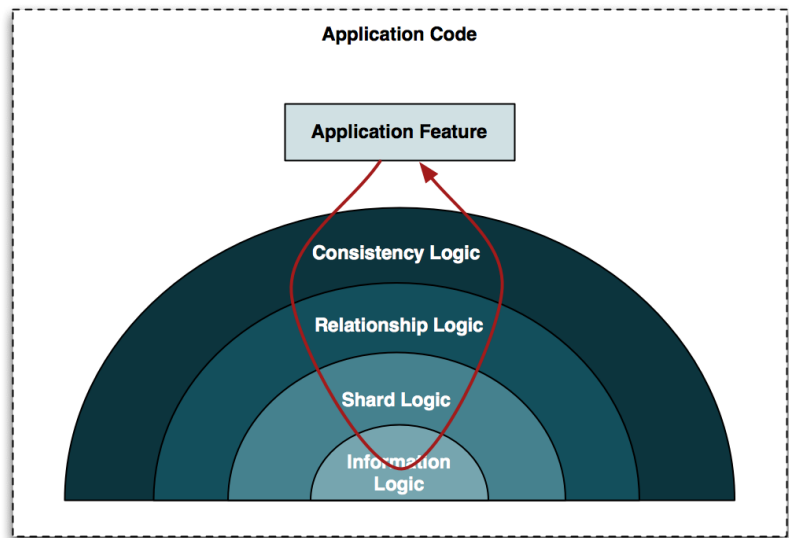
From this point forward, any amount of more complex relational logic (perhaps we want to add buyer feedback, which requires new sharded tables) or analytics (how many objects of what type have been sold) becomes at least 3x harder. That means more time spent, more bugs created, more QA time to resolve, and more unhappy customers. As a developer, adding any new feature becomes a major undertaking. The world of feature development used to look like Figure 5 to the right.

FIGURE 5



The application feature used to be able to deal directly with the information logic without explicit reference to anything other than the SQL schema. After sharding, feature development looks like Figure 6

FIGURE 6



## The “Deep Freeze” Way of Life

Every feature becomes a major undertaking because you have to dig through all of the coding layers related to consistency between shards, relationship layouts between shards, the basic logic for choosing and connecting to a shard, before you even get to the information logic. Application development becomes needlessly complex and difficult. All of these extra layers in the application aren't adding any

particular value to the application other than the ability to work around the lack of legacy databases' ability to scale out. This is the path to the sharding “deep freeze” where development pain is a way of life.

Solving this problem once and for all is why we started Clustrix.

## ABOUT CLUSTRIX, INC.

Clustrix is the leader in NewSQL databases for transactional big-data applications. It enables fast-growing online businesses to rapidly scale to unlimited users, transactions and data, with no database sharding and full ACID compliance. Clustrix is delivered as an optimized appliance that is easy to install and automates fault tolerance as the database grows.

Clustrix helps break the vicious cycle of database cost and complexity with a simpler, more elegant approach that allows you to focus 100% on innovation. A unique parallel query model enables linear transactional performance as you add nodes. Start fast, scale fast, grow big and never hit the wall.

## CONTACT US

TEL: 415-501-9560

EMAIL: [info@clustrix.com](mailto:info@clustrix.com)

WEB: [www.clustrix.com](http://www.clustrix.com)

Clustrix, Inc.

201 Mission Street, Suite 800

San Francisco, California 94105