Chapter 2. Aggregate Data Models

A data model is the model through which we perceive and manipulate our data. For people using a database, the data model describes how we interact with the data in the database. This is distinct from a storage model, which describes how the database stores and manipulates the data internally. In an ideal world, we should be ignorant of the storage model, but in practice we need at least some inkling of it—primarily to achieve decent performance.

In conversation, the term "data model" often means the model of the specific data in an application. A developer might point to an entity-relationship diagram of their database and refer to that as their data model containing customers, orders, products, and the like. However, in this book we'll mostly be using "data model" to refer to the model by which the database organizes data—what might be more formally called a metamodel.

The dominant data model of the last couple of decades is the relational data model, which is best visualized as a set of tables, rather like a page of a spreadsheet. Each table has rows, with each row representing some entity of interest. We describe this entity through columns, each having a single value. A column may refer to another row in the same or different table, which constitutes a relationship between those entities. (We're using informal but common terminology when we speak of tables and rows; the more formal terms would be relations and tuples.)

One of the most obvious shifts with NoSQL is a move away from the relational model. Each NoSQL solution has a different model that it uses, which we put into four categories widely used in the NoSQL ecosystem: key-value, document, column-family, and graph. Of these, the first three share a common characteristic of their data models which we will call aggregate orientation. In this chapter we'll explain what we mean by aggregate orientation and what it means for data models.

2.1. Aggregates

The relational model takes the information that we want to store and divides it into tuples (rows). A tuple is a limited data structure: It captures a set of values, so you cannot nest one tuple within another to get nested records, nor can you put a list of values or tuples within another. This simplicity underpins the relational model—it allows us to think of all operations as operating on and returning tuples.

Aggregate orientation takes a different approach. It recognizes that often, you want to operate on data in units that have a more complex structure than a set of tuples. It can be handy to think in terms of a complex record that allows lists and other record structures to be nested inside it. As we'll see, key-value, document, and column-family databases all make use of this more complex record. However, there is no common term for this complex record; in this book we use the term "aggregate."

Aggregate is a term that comes from Domain-Driven Design [Evans]. In Domain-Driven Design, an **aggregate** is a collection of related objects that we wish to treat as a unit. In particular, it is a unit for data manipulation and management of consistency. Typically, we like to update aggregates with atomic operations and communicate with our data storage in terms of aggregates. This definition matches really well with how key-value, document, and column-family databases work. Dealing in aggregates makes it much easier for these databases to handle operating on a cluster, since the aggregate makes a natural unit for replication and sharding. Aggregates are also often easier for application programmers to work with, since they often manipulate data through aggregate structures.

2.1.1. Example of Relations and Aggregates

At this point, an example may help explain what we're talking about. Let's assume we have to build an e-commerce website; we are going to be selling items directly to customers over the web, and we will have to store information about users, our product catalog, orders, shipping addresses, billing addresses, and payment data. We can use this scenario to model the data using a relation data store as well as NoSQL data stores and talk about their pros and cons. For a relational database, we might start with a data model shown in Figure 2.1.

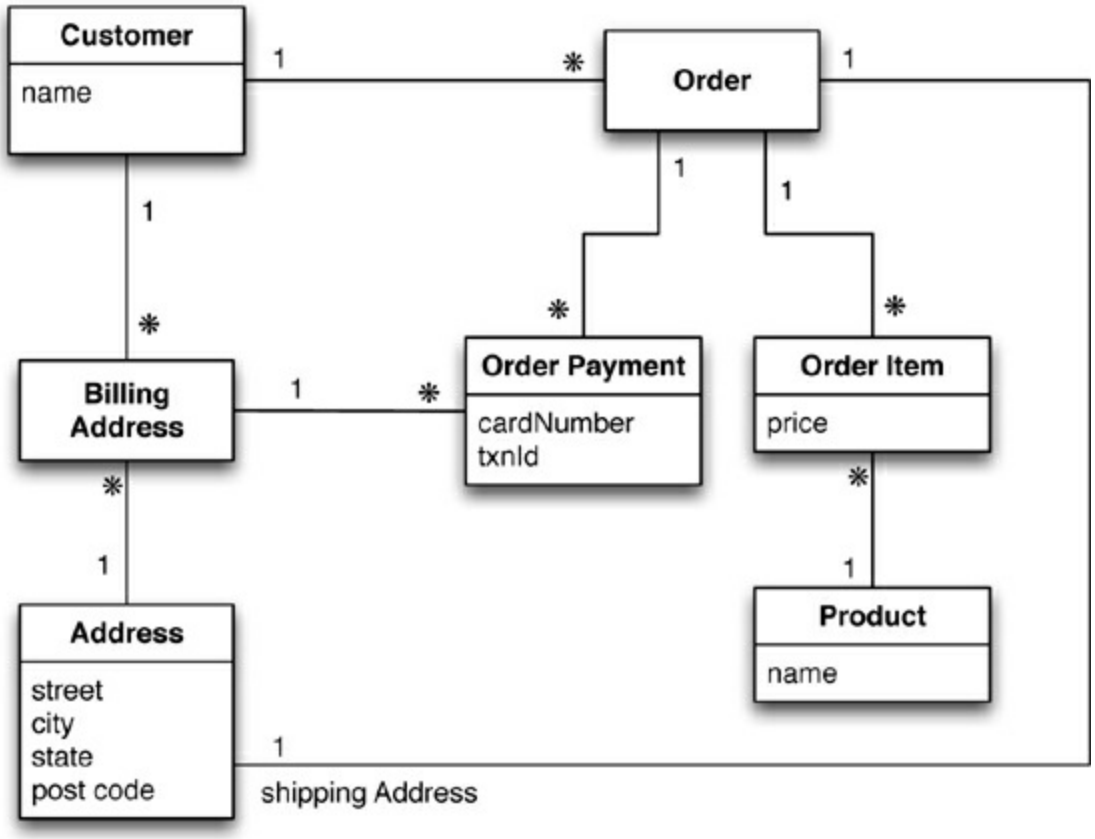

Figure 2.1. Data model oriented around a relational database (using UML notation [Fowler UML])

Figure 2.2 presents some sample data for this model.

Customer	
Id	Name
1	Martin

Orders		
Id	CustomerId	ShippingAddressId
99	1	77

Product	
Id	Name
27	NoSQL Distilled

BillingAddress		
Id	CustomerId	AddressId
55	1	77

OrderItem			
Id	OrderId	ProductId	Price
100	99	27	32.45

Address	
Id	City
77	Chicago

OrderPayment				
Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abelif879rft

Figure 2.2. Typical data using RDBMS data model

As we're good relational soldiers, everything is properly normalized, so that no data is repeated in multiple tables. We also have referential integrity. A realistic order system would naturally be more involved than this, but this is the benefit of the rarefied air of a book.

Now let's see how this model might look when we think in more aggregate-oriented terms (Figure 2.3).

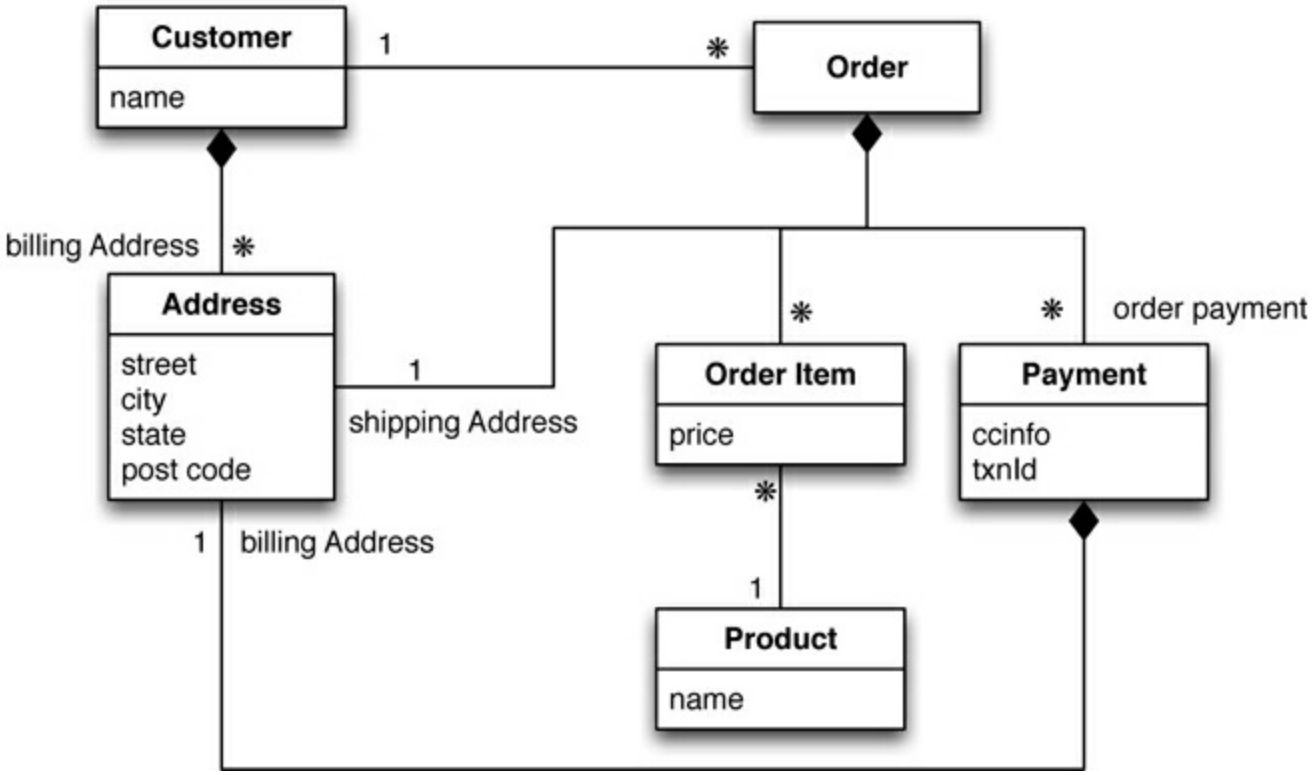

Figure 2.3. An aggregate data model

Again, we have some sample data, which we'll show in JSON format as that's a common

representation for data in NoSQL land.

Click here to view code image

```
// in customers
{
"id":1,
"name":"Martin",
"billingAddress":[{"city":"Chicago"}]
}

// in orders
{
"id":99,
"customerId":1,
"orderItems":[
  {
  "productId":27,
  "price": 32.45,
  "productName": "NoSQL Distilled"
    }
  ],
"shippingAddress":[{"city":"Chicago"}]
"orderPayment":[
  {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
  }
  ],
}
```

In this model, we have two main aggregates: customer and order. We've used the black-diamond composition marker in UML to show how data fits into the aggregation structure. The customer contains a list of billing addresses; the order contains a list of order items, a shipping address, and payments. The payment itself contains a billing address for that payment.

A single logical address record appears three times in the example data, but instead of using IDs it's treated as a value and copied each time. This fits the domain where we would not want the shipping address, nor the payment's billing address, to change. In a relational database, we would ensure that the address rows aren't updated for this case, making a new row instead. With aggregates, we can copy the whole address structure into the aggregate as we need to.

The link between the customer and the order isn't within either aggregate—it's a relationship between aggregates. Similarly, the link from an order item would cross into a separate aggregate structure for products, which we haven't gone into. We've shown the product name as part of the order item here—this kind of denormalization is similar to the tradeoffs with relational databases, but is more common with aggregates because we want to minimize the number of aggregates we access during a data interaction.

The important thing to notice here isn't the particular way we've drawn the aggregate boundary so much as the fact that you have to think about accessing that data—and make that part of your thinking when developing the application data model. Indeed we could draw our aggregate boundaries differently, putting all the orders for a customer into the customer aggregate (Figure 2.4).

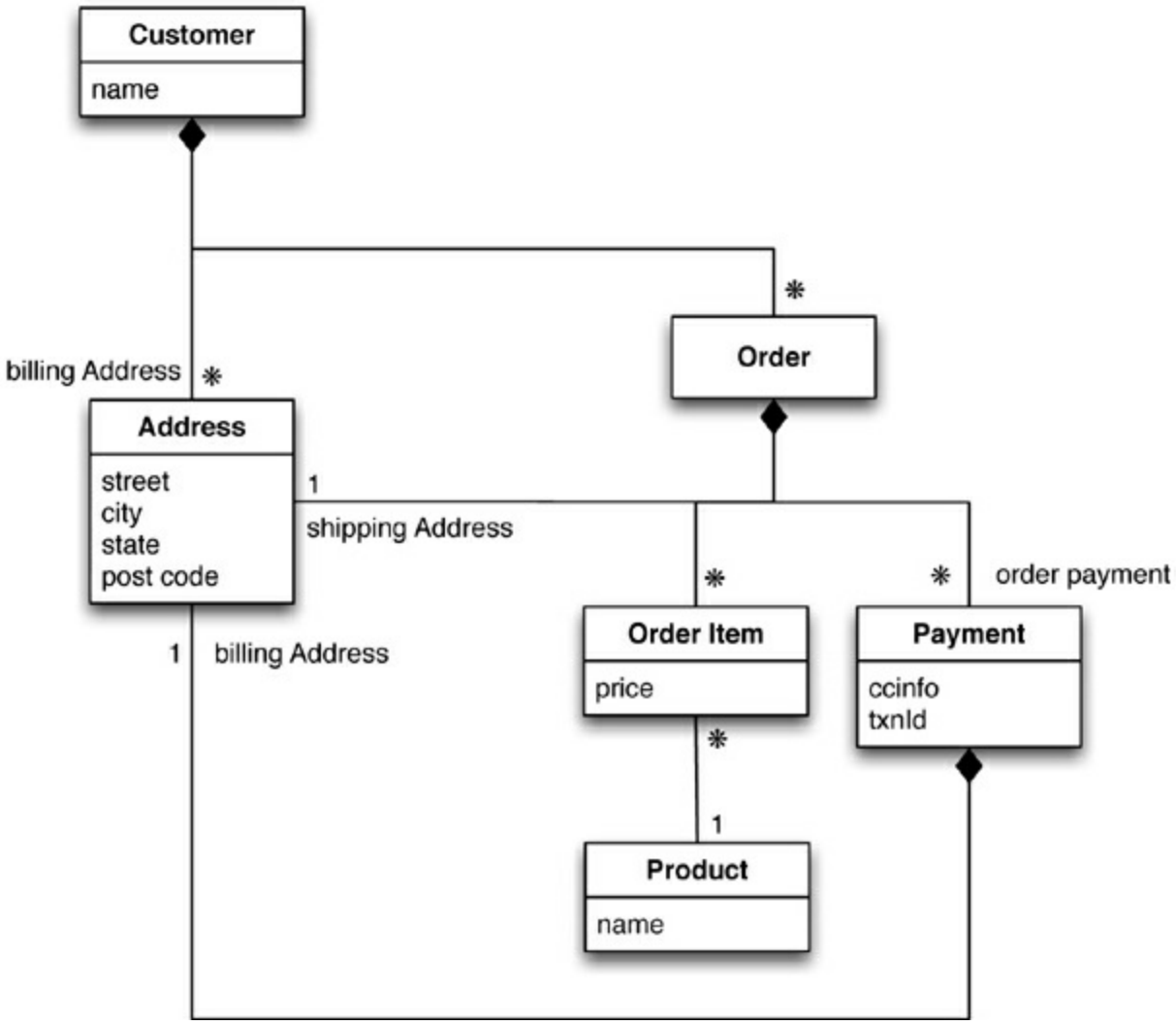

Figure 2.4. Embed all the objects for customer and the customer's orders

Using the above data model, an example `Customer` and `Order` would look like this:

Click here to view code image

```
// in customers
{
"customer": {
"id": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"orders": [
  {
    "id":99,
    "customerId":1,
    "orderItems":[
    {
    "productId":27,
    "price": 32.45,
    "productName": "NoSQL Distilled"
    }
  ],
  "shippingAddress":[{"city":"Chicago"}]
  "orderPayment":[
    {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
    }],
```

```
  }]
}
}
```

Like most things in modeling, there's no universal answer for how to draw your aggregate boundaries. It depends entirely on how you tend to manipulate your data. If you tend to access a customer together with all of that customer's orders at once, then you would prefer a single aggregate. However, if you tend to focus on accessing a single order at a time, then you should prefer having separate aggregates for each order. Naturally, this is very context-specific; some applications will prefer one or the other, even within a single system, which is exactly why many people prefer aggregate ignorance.

2.1.2. Consequences of Aggregate Orientation

While the relational mapping captures the various data elements and their relationships reasonably well, it does so without any notion of an aggregate entity. In our domain language, we might say that an order consists of order items, a shipping address, and a payment. This can be expressed in the relational model in terms of foreign key relationships—but there is nothing to distinguish relationships that represent aggregations from those that don't. As a result, the database can't use a knowledge of aggregate structure to help it store and distribute the data.

Various data modeling techniques have provided ways of marking aggregate or composite structures. The problem, however, is that modelers rarely provide any semantics for what makes an aggregate relationship different from any other; where there are semantics, they vary. When working with aggregate-oriented databases, we have a clearer semantics to consider by focusing on the unit of interaction with the data storage. It is, however, not a logical data property: It's all about how the data is being used by applications—a concern that is often outside the bounds of data modeling.

Relational databases have no concept of aggregate within their data model, so we call them **aggregate-ignorant**. In the NoSQL world, graph databases are also aggregate-ignorant. Being aggregate-ignorant is not a bad thing. It's often difficult to draw aggregate boundaries well, particularly if the same data is used in many different contexts. An order makes a good aggregate when a customer is making and reviewing orders, and when the retailer is processing orders. However, if a retailer wants to analyze its product sales over the last few months, then an order aggregate becomes a trouble. To get to product sales history, you'll have to dig into every aggregate in the database. So an aggregate structure may help with some data interactions but be an obstacle for others. An aggregate-ignorant model allows you to easily look at the data in different ways, so it is a better choice when you don't have a primary structure for manipulating your data.

The clinching reason for aggregate orientation is that it helps greatly with running on a cluster, which as you'll remember is the killer argument for the rise of NoSQL. If we're running on a cluster, we need to minimize how many nodes we need to query when we are gathering data. By explicitly including aggregates, we give the database important information about which bits of data will be manipulated together, and thus should live on the same node.

Aggregates have an important consequence for transactions. Relational databases allow you to manipulate any combination of rows from any tables in a single transaction. Such transactions are called **ACID transactions**: Atomic, Consistent, Isolated, and Durable. ACID is a rather contrived acronym; the real point is the atomicity: Many rows spanning many tables are updated as a single operation. This operation either succeeds or fails in its entirety, and concurrent operations are

isolated from each other so they cannot see a partial update.

It's often said that NoSQL databases don't support ACID transactions and thus sacrifice consistency. This is a rather sweeping simplification. In general, it's true that aggregate-oriented databases don't have ACID transactions that span multiple aggregates. Instead, they support atomic manipulation of a single aggregate at a time. This means that if we need to manipulate multiple aggregates in an atomic way, we have to manage that ourselves in the application code. In practice, we find that most of the time we are able to keep our atomicity needs to within a single aggregate; indeed, that's part of the consideration for deciding how to divide up our data into aggregates. We should also remember that graph and other aggregate-ignorant databases usually do support ACID transactions similar to relational databases. Above all, the topic of consistency is much more involved than whether a database is ACID or not, as we'll explore in Chapter 5.

2.2. Key-Value and Document Data Models

We said earlier on that key-value and document databases were strongly aggregate-oriented. What we meant by this was that we think of these databases as primarily constructed through aggregates. Both of these types of databases consist of lots of aggregates with each aggregate having a key or ID that's used to get at the data.

The two models differ in that in a key-value database, the aggregate is opaque to the database—just some big blob of mostly meaningless bits. In contrast, a document database is able to see a structure in the aggregate. The advantage of opacity is that we can store whatever we like in the aggregate. The database may impose some general size limit, but other than that we have complete freedom. A document database imposes limits on what we can place in it, defining allowable structures and types. In return, however, we get more flexibility in access.

With a key-value store, we can only access an aggregate by lookup based on its key. With a document database, we can submit queries to the database based on the fields in the aggregate, we can retrieve part of the aggregate rather than the whole thing, and database can create indexes based on the contents of the aggregate.

In practice, the line between key-value and document gets a bit blurry. People often put an ID field in a document database to do a key-value style lookup. Databases classified as key-value databases may allow you structures for data beyond just an opaque aggregate. For example, Riak allows you to add metadata to aggregates for indexing and interaggregate links, Redis allows you to break down the aggregate into lists or sets. You can support querying by integrating search tools such as Solr. As an example, Riak includes a search facility that uses Solr-like searching on any aggregates that are stored as JSON or XML structures.

Despite this blurriness, the general distinction still holds. With key-value databases, we expect to mostly look up aggregates using a key. With document databases, we mostly expect to submit some form of query based on the internal structure of the document; this might be a key, but it's more likely to be something else.

2.3. Column-Family Stores

One of the early and influential NoSQL databases was Google's BigTable [Chang etc.]. Its name conjured up a tabular structure which it realized with sparse columns and no schema. As you'll soon see, it doesn't help to think of this structure as a table; rather, it is a two-level map. But, however you think about the structure, it has been a model that influenced later databases such as HBase and

Chapter 4. Distribution Models

The primary driver of interest in NoSQL has been its ability to run databases on a large cluster. As data volumes increase, it becomes more difficult and expensive to scale up—buy a bigger server to run the database on. A more appealing option is to scale out—run the database on a cluster of servers. Aggregate orientation fits well with scaling out because the aggregate is a natural unit to use for distribution.

Depending on your distribution model, you can get a data store that will give you the ability to handle larger quantities of data, the ability to process a greater read or write traffic, or more availability in the face of network slowdowns or breakages. These are often important benefits, but they come at a cost. Running over a cluster introduces complexity—so it's not something to do unless the benefits are compelling.

Broadly, there are two paths to data distribution: replication and sharding. Replication takes the same data and copies it over multiple nodes. Sharding puts different data on different nodes. Replication and sharding are orthogonal techniques: You can use either or both of them. Replication comes into two forms: master-slave and peer-to-peer. We will now discuss these techniques starting at the simplest and working up to the more complex: first single-server, then master-slave replication, then sharding, and finally peer-to-peer replication.

4.1. Single Server

The first and the simplest distribution option is the one we would most often recommend—no distribution at all. Run the database on a single machine that handles all the reads and writes to the data store. We prefer this option because it eliminates all the complexities that the other options introduce; it's easy for operations people to manage and easy for application developers to reason about.

Although a lot of NoSQL databases are designed around the idea of running on a cluster, it can make sense to use NoSQL with a single-server distribution model if the data model of the NoSQL store is more suited to the application. Graph databases are the obvious category here—these work best in a single-server configuration. If your data usage is mostly about processing aggregates, then a single-server document or key-value store may well be worthwhile because it's easier on application developers.

For the rest of this chapter we'll be wading through the advantages and complications of more sophisticated distribution schemes. Don't let the volume of words fool you into thinking that we would prefer these options. If we can get away without distributing our data, we will always choose a single-server approach.

4.2. Sharding

Often, a busy data store is busy because different people are accessing different parts of the dataset. In these circumstances we can support horizontal scalability by putting different parts of the data onto different servers—a technique that's called **sharding** (see Figure 4.1).

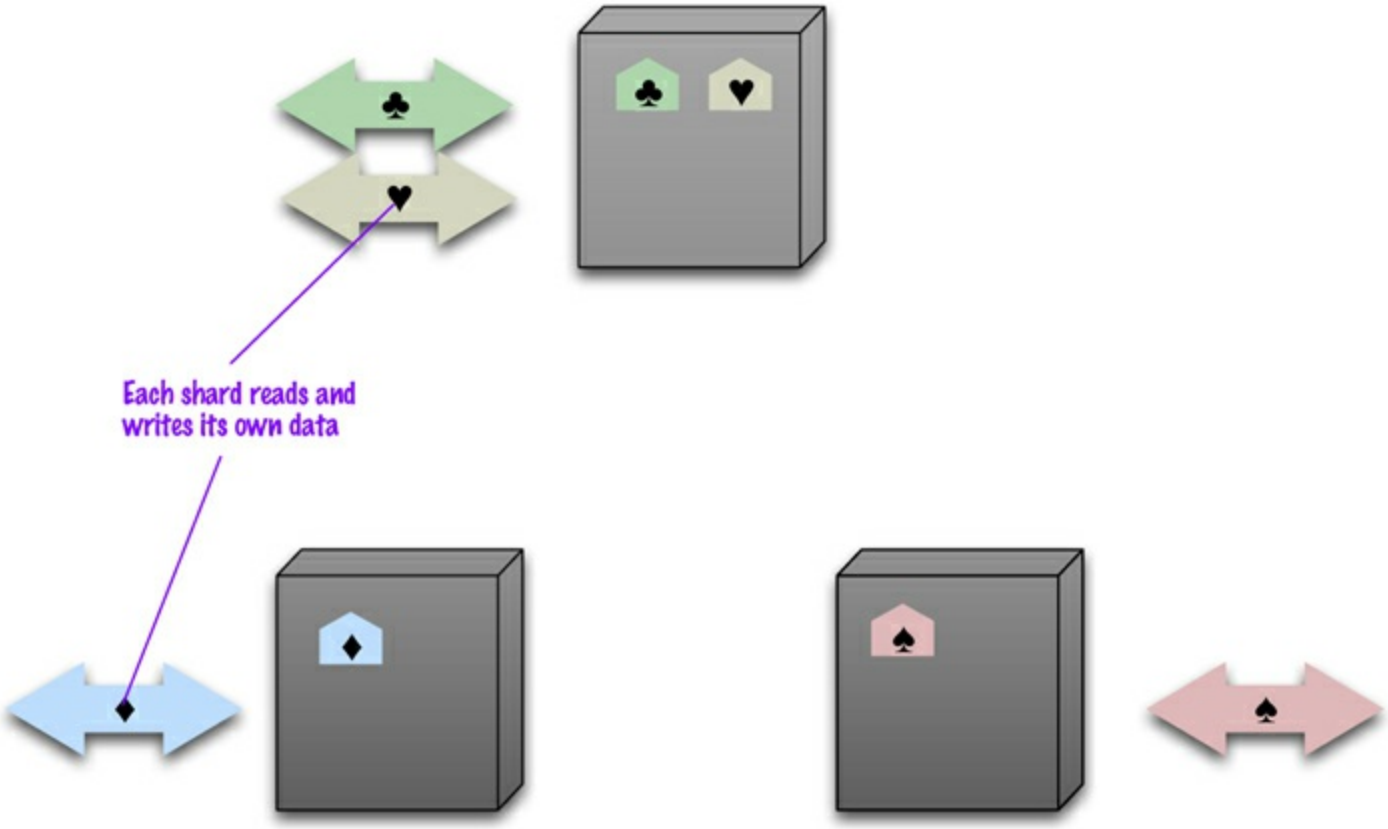

Figure 4.1. Sharding puts different data on separate nodes, each of which does its own reads and writes.

In the ideal case, we have different users all talking to different server nodes. Each user only has to talk to one server, so gets rapid responses from that server. The load is balanced out nicely between servers—for example, if we have ten servers, each one only has to handle 10% of the load.

Of course the ideal case is a pretty rare beast. In order to get close to it we have to ensure that data that's accessed together is clumped together on the same node and that these clumps are arranged on the nodes to provide the best data access.

The first part of this question is how to clump the data up so that one user mostly gets her data from a single server. This is where aggregate orientation comes in really handy. The whole point of aggregates is that we design them to combine data that's commonly accessed together—so aggregates leap out as an obvious unit of distribution.

When it comes to arranging the data on the nodes, there are several factors that can help improve performance. If you know that most accesses of certain aggregates are based on a physical location, you can place the data close to where it's being accessed. If you have orders for someone who lives in Boston, you can place that data in your eastern US data center.

Another factor is trying to keep the load even. This means that you should try to arrange aggregates so they are evenly distributed across the nodes which all get equal amounts of the load. This may vary over time, for example if some data tends to be accessed on certain days of the week—so there may be domain-specific rules you'd like to use.

In some cases, it's useful to put aggregates together if you think they may be read in sequence. The Bigtable paper [Chang etc.] described keeping its rows in lexicographic order and sorting web addresses based on reversed domain names (e.g., `com.martinfowler`). This way data for multiple pages could be accessed together to improve processing efficiency.

Historically most people have done sharding as part of application logic. You might put all

customers with surnames starting from A to D on one shard and E to G on another. This complicates the programming model, as application code needs to ensure that queries are distributed across the various shards. Furthermore, rebalancing the sharding means changing the application code and migrating the data. Many NoSQL databases offer **auto-sharding**, where the database takes on the responsibility of allocating data to shards and ensuring that data access goes to the right shard. This can make it much easier to use sharding in an application.

Sharding is particularly valuable for performance because it can improve both read and write performance. Using replication, particularly with caching, can greatly improve read performance but does little for applications that have a lot of writes. Sharding provides a way to horizontally scale writes.

Sharding does little to improve resilience when used alone. Although the data is on different nodes, a node failure makes that shard's data unavailable just as surely as it does for a single-server solution. The resilience benefit it does provide is that only the users of the data on that shard will suffer; however, it's not good to have a database with part of its data missing. With a single server it's easier to pay the effort and cost to keep that server up and running; clusters usually try to use less reliable machines, and you're more likely to get a node failure. So in practice, sharding alone is likely to decrease resilience.

Despite the fact that sharding is made much easier with aggregates, it's still not a step to be taken lightly. Some databases are intended from the beginning to use sharding, in which case it's wise to run them on a cluster from the very beginning of development, and certainly in production. Other databases use sharding as a deliberate step up from a single-server configuration, in which case it's best to start single-server and only use sharding once your load projections clearly indicate that you are running out of headroom.

In any case the step from a single node to sharding is going to be tricky. We have heard tales of teams getting into trouble because they left sharding to very late, so when they turned it on in production their database became essentially unavailable because the sharding support consumed all the database resources for moving the data onto new shards. The lesson here is to use sharding well before you need to—when you have enough headroom to carry out the sharding.

4.3. Master-Slave Replication

With master-slave distribution, you replicate data across multiple nodes. One node is designated as the master, or primary. This master is the authoritative source for the data and is usually responsible for processing any updates to that data. The other nodes are slaves, or secondaries. A replication process synchronizes the slaves with the master (see Figure 4.2).

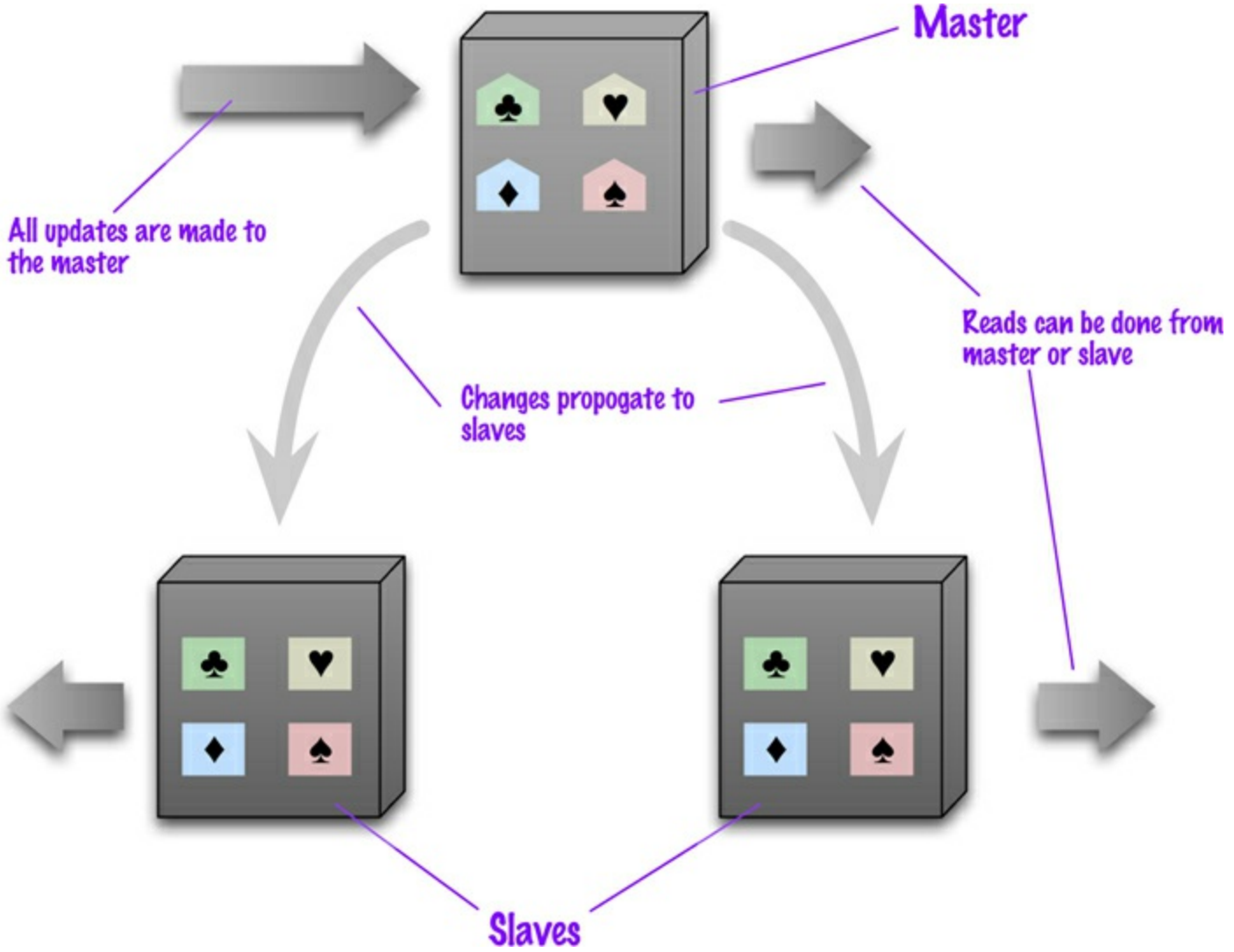

Figure 4.2. Data is replicated from master to slaves. The master services all writes; reads may come from either master or slaves.

Master-slave replication is most helpful for scaling when you have a read-intensive dataset. You can scale horizontally to handle more read requests by adding more slave nodes and ensuring that all read requests are routed to the slaves. You are still, however, limited by the ability of the master to process updates and its ability to pass those updates on. Consequently it isn't such a good scheme for datasets with heavy write traffic, although offloading the read traffic will help a bit with handling the write load.

A second advantage of master-slave replication is **read resilience**: Should the master fail, the slaves can still handle read requests. Again, this is useful if most of your data access is reads. The failure of the master does eliminate the ability to handle writes until either the master is restored or a new master is appointed. However, having slaves as replicates of the master does speed up recovery after a failure of the master since a slave can be appointed a new master very quickly.

The ability to appoint a slave to replace a failed master means that master-slave replication is useful even if you don't need to scale out. All read and write traffic can go to the master while the slave acts as a hot backup. In this case it's easiest to think of the system as a single-server store with a hot backup. You get the convenience of the single-server configuration but with greater resilience—which is particularly handy if you want to be able to handle server failures gracefully.

Masters can be appointed manually or automatically. Manual appointing typically means that when you configure your cluster, you configure one node as the master. With automatic appointment, you

create a cluster of nodes and they elect one of themselves to be the master. Apart from simpler configuration, automatic appointment means that the cluster can automatically appoint a new master when a master fails, reducing downtime.

In order to get read resilience, you need to ensure that the read and write paths into your application are different, so that you can handle a failure in the write path and still read. This includes such things as putting the reads and writes through separate database connections—a facility that is not often supported by database interaction libraries. As with any feature, you cannot be sure you have read resilience without good tests that disable the writes and check that reads still occur.

Replication comes with some alluring benefits, but it also comes with an inevitable dark side—inconsistency. You have the danger that different clients, reading different slaves, will see different values because the changes haven't all propagated to the slaves. In the worst case, that can mean that a client cannot read a write it just made. Even if you use master-slave replication just for hot backup this can be a concern, because if the master fails, any updates not passed on to the backup are lost. We'll talk about how to deal with these issues later ("Consistency," p. 47).

4.4. Peer-to-Peer Replication

Master-slave replication helps with read scalability but doesn't help with scalability of writes. It provides resilience against failure of a slave, but not of a master. Essentially, the master is still a bottleneck and a single point of failure. Peer-to-peer replication (see Figure 4.3) attacks these problems by not having a master. All the replicas have equal weight, they can all accept writes, and the loss of any of them doesn't prevent access to the data store.

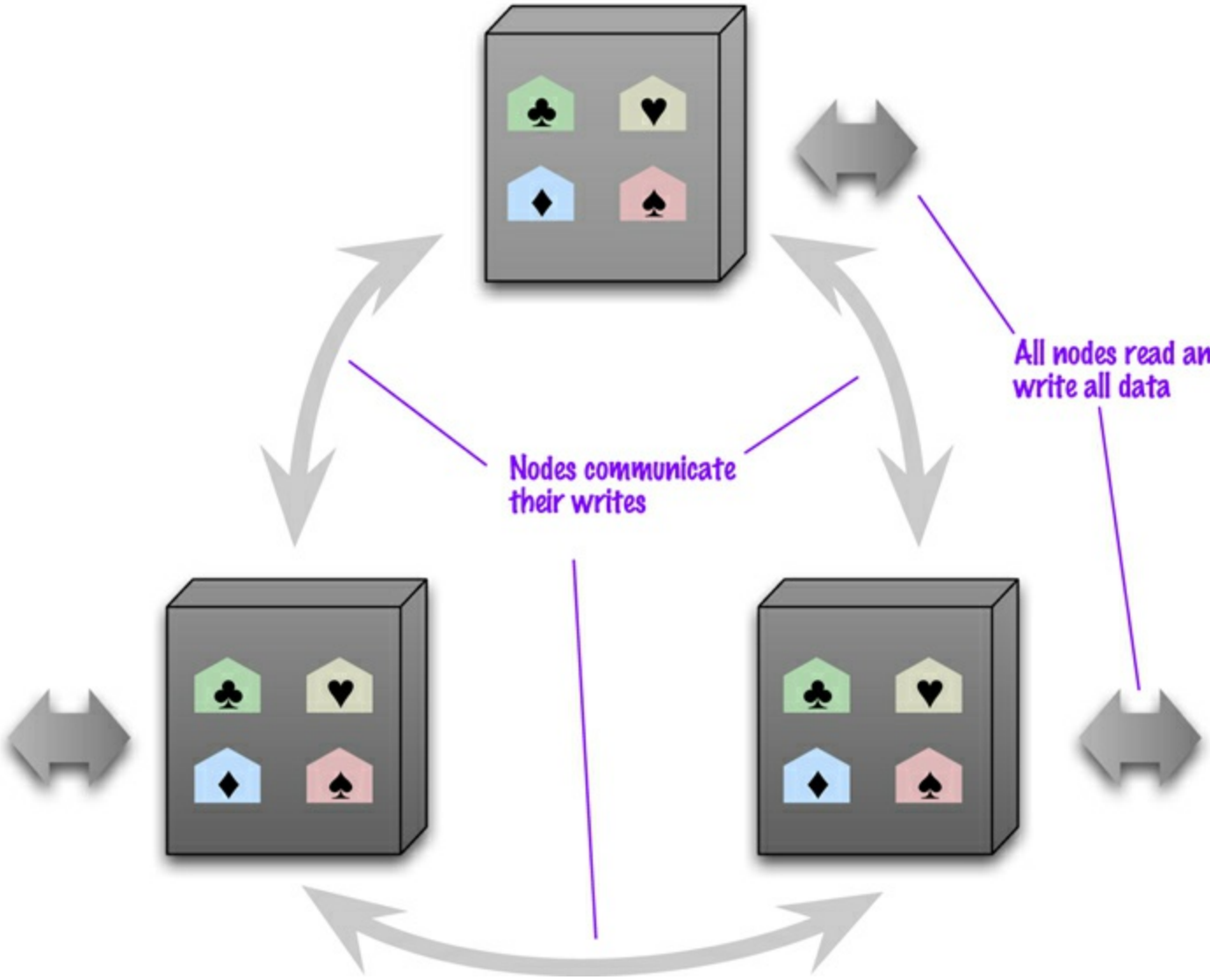

Figure 4.3. Peer-to-peer replication has all nodes applying reads and writes to all the data.

The prospect here looks mighty fine. With a peer-to-peer replication cluster, you can ride over node failures without losing access to data. Furthermore, you can easily add nodes to improve your performance. There's much to like here—but there are complications.

The biggest complication is, again, consistency. When you can write to two different places, you run the risk that two people will attempt to update the same record at the same time—a write-write conflict. Inconsistencies on read lead to problems but at least they are relatively transient. Inconsistent writes are forever.

We'll talk more about how to deal with write inconsistencies later on, but for the moment we'll note a couple of broad options. At one end, we can ensure that whenever we write data, the replicas coordinate to ensure we avoid a conflict. This can give us just as strong a guarantee as a master, albeit at the cost of network traffic to coordinate the writes. We don't need all the replicas to agree on the write, just a majority, so we can still survive losing a minority of the replica nodes.

At the other extreme, we can decide to cope with an inconsistent write. There are contexts when we can come up with policy to merge inconsistent writes. In this case we can get the full performance benefit of writing to any replica.

These points are at the ends of a spectrum where we trade off consistency for availability.

4.5. Combining Sharding and Replication

Replication and sharding are strategies that can be combined. If we use both master-slave replication and sharding (see Figure 4.4), this means that we have multiple masters, but each data item only has a single master. Depending on your configuration, you may choose a node to be a master for some data and slaves for others, or you may dedicate nodes for master or slave duties.

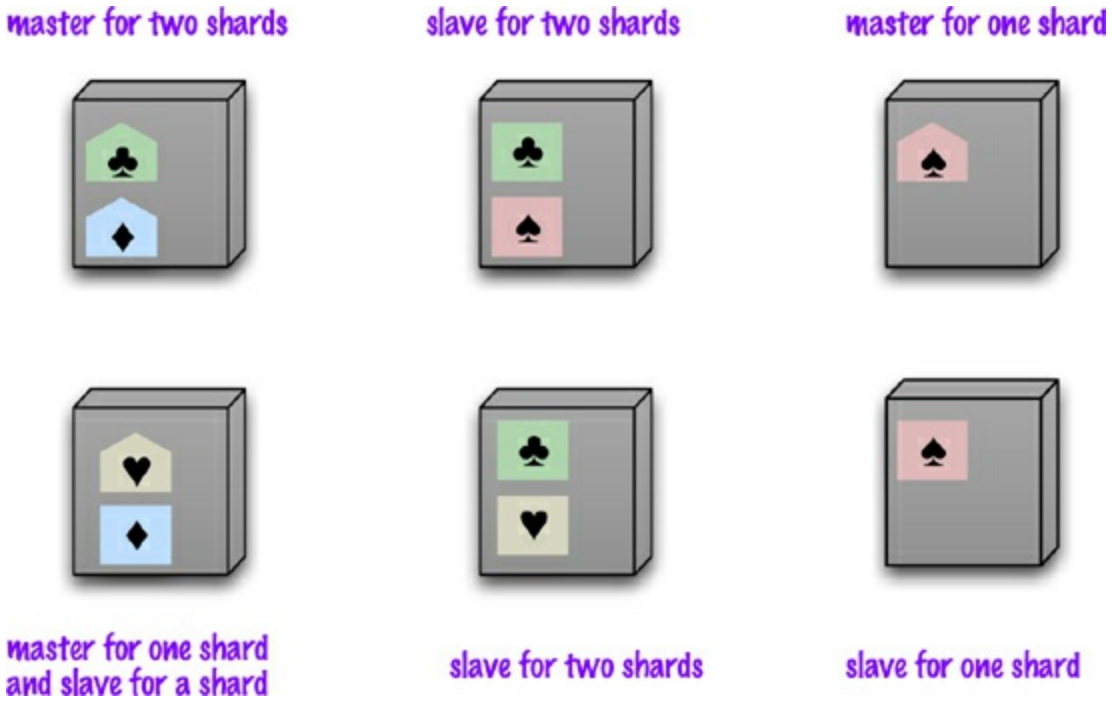

Figure 4.4. Using master-slave replication together with sharding

Using peer-to-peer replication and sharding is a common strategy for column-family databases. In a scenario like this you might have tens or hundreds of nodes in a cluster with data sharded over them. A good starting point for peer-to-peer replication is to have a replication factor of 3, so each shard is present on three nodes. Should a node fail, then the shards on that node will be built on the other nodes (see Figure 4.5).

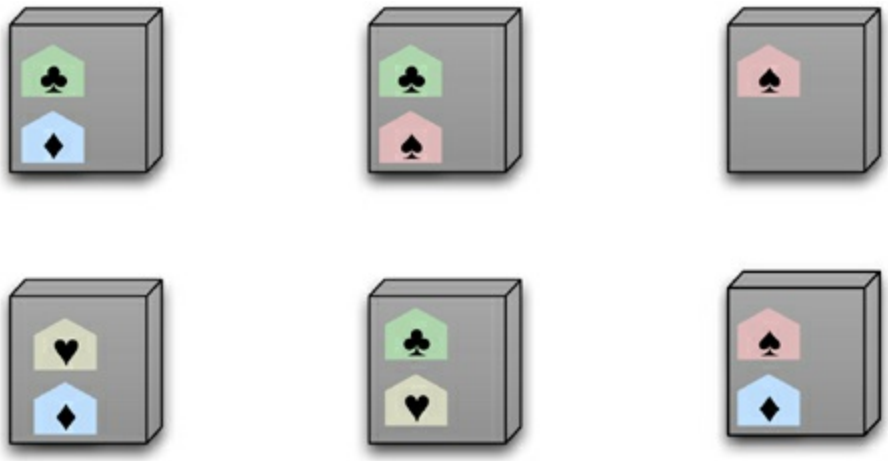

Figure 4.5. Using peer-to-peer replication together with sharding

4.6. Key Points

- There are two styles of distributing data:
  - Sharding distributes different data across multiple servers, so each server acts as the single source for a subset of data.
  - Replication copies data across multiple servers, so each bit of data can be found in multiple places.

A system may use either or both techniques.

- Replication comes in two forms:
  - Master-slave replication makes one node the authoritative copy that handles writes while slaves synchronize with the master and may handle reads.
  - Peer-to-peer replication allows writes to any node; the nodes coordinate to synchronize their copies of the data.

  Master-slave replication reduces the chance of update conflicts but peer-to-peer replication avoids loading all writes onto a single point of failure.