style.

1.6. Key Points

- Relational databases have been a successful technology for twenty years, providing persistence, concurrency control, and an integration mechanism.
- Application developers have been frustrated with the impedance mismatch between the relational model and the in-memory data structures.
- There is a movement away from using databases as integration points towards encapsulating databases within applications and integrating through services.
- The vital factor for a change in data storage was the need to support large volumes of data by running on clusters. Relational databases are not designed to run efficiently on clusters.
- NoSQL is an accidental neologism. There is no prescriptive definition—all you can make is an observation of common characteristics.
- The common characteristics of NoSQL databases are
  - Not using the relational model
  - Running well on clusters
  - Open-source
  - Built for the 21st century web estates
  - Schemaless
- The most important result of the rise of NoSQL is Polyglot Persistence.

AGGREGATE DATA MODELS:

→ a data model (→ meta 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.

⇒ 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.

→ 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.

→ Aggregate is a term that comes from Domain-Driven Design (DDD)

→ In DDD, 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 consitency.

→ typically, we like to update aggregates with atomic operations and communicate with our data storage in terms of aggregates.

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.

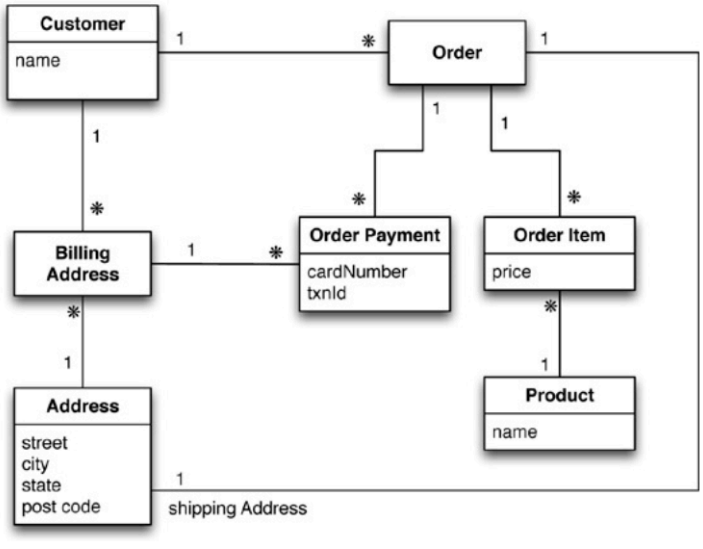

Figure 2.1. Data model oriented around a relational database (using UML notation [Fowler UML])

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).

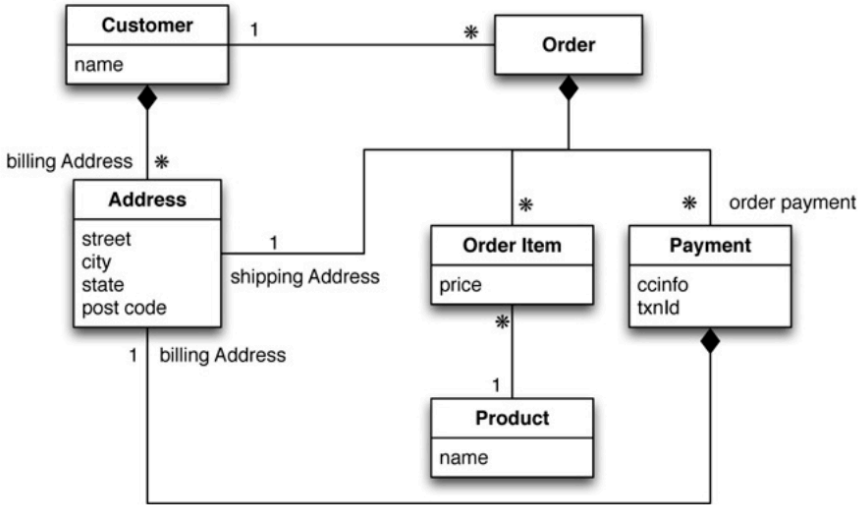

Figure 2.3. An aggregate data model

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).

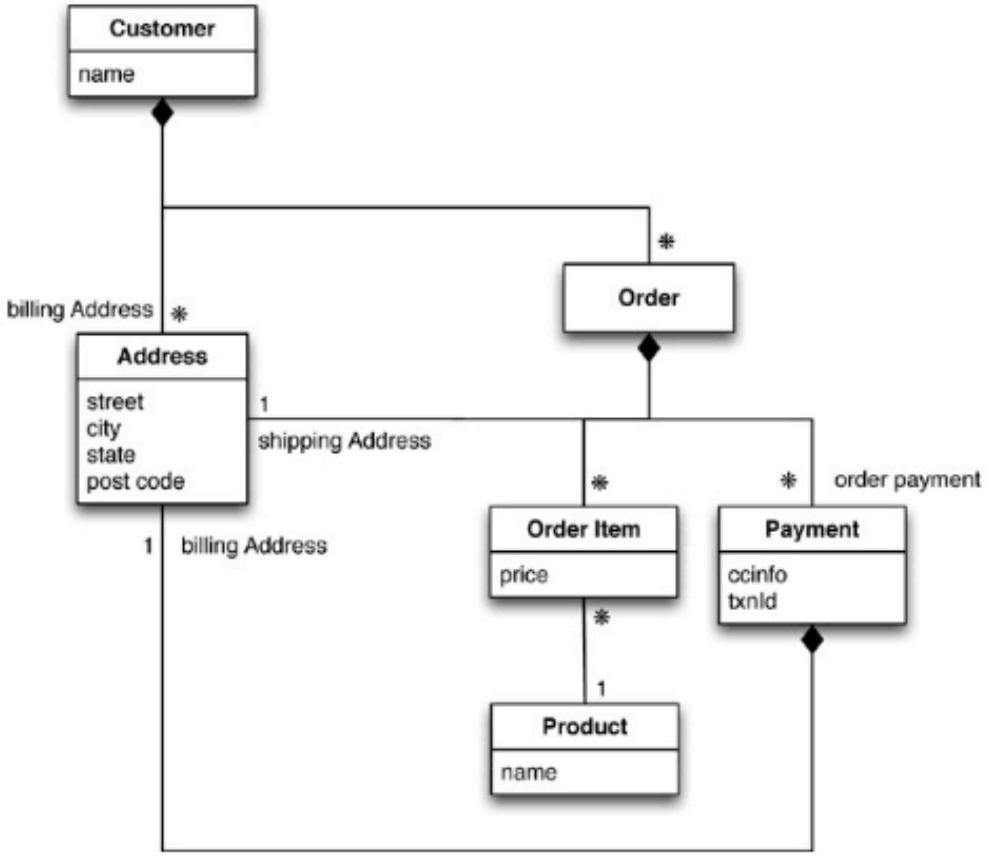

Figure 2.4. Embed all the objects for customer and the customer's orders

```
// 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.

⇒ Consequences of Aggregate Orientation:

1. Relational databases store related data using tables and foreign keys but do not recognize them as a single unit (aggregate).
2. An aggregate is a group of related data (like order, items, payment) that is usually used together in applications.
3. Traditional data models may show composite structures but lack clear, consistent semantics for aggregates.
4. Aggregate-oriented databases define aggregates based on how applications read and write data.
5. Aggregates are about **usage patterns**, not logical data structure.
6. Databases that do not understand aggregates are called **aggregate-ignorant**.
7. Relational and graph databases are aggregate-ignorant but this is not a disadvantage by itself.
8. Aggregates work well when the same data is frequently accessed together (e.g., order creation and processing).
9. Aggregates become inefficient for analytical queries that need to scan data across many aggregates.
10. Aggregate-ignorant models allow flexible querying from multiple perspectives.
11. The main reason for aggregate orientation is efficient operation in **distributed clusters**.
12. Aggregates help ensure related data is stored on the same node, reducing cross-node queries.
13. Relational databases support **ACID transactions** across multiple tables and rows.
14. ACID ensures atomic updates where operations fully succeed or fully fail.
15. Aggregate-oriented NoSQL databases support atomic operations only within a **single aggregate**.
16. Multi-aggregate atomic updates must be handled by application logic.
17. In practice, most operations fit within one aggregate, guiding aggregate design.
18. Graph and aggregate-ignorant databases usually support full ACID transactions.
19. Consistency is more complex than just ACID vs NoSQL and depends on system design choices.

⇒ Key-value and Document Data Models:

→ they are strongly aggregate oriented

→ both of these types of databases consist of lots of aggregates with each aggregate having a key or ID that is used to get at the data.

→ the 2 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 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.

Aspect	Key-Value Database	Document Database
Aggregate Orientation	Strongly aggregate-oriented	Strongly aggregate-oriented
Basic Unit of Storage	Aggregate stored as a value	Aggregate stored as a document
Identifier	Accessed using a unique key/ID	Accessed using key or document fields
Visibility of Structure	Aggregate is **opaque** (database can't see inside)	Aggregate structure is **visible**
Data Flexibility	Can store anything as a blob	Must follow supported document structure
Schema Constraints	Very minimal (mostly size limits)	Enforces data types and structure
Query Capability	Only key-based lookup	Field-based queries supported
Partial Retrieval	Not supported (whole aggregate fetched)	Supported (fetch part of document)
Indexing	Generally limited or external	Indexes can be created on fields
Metadata Support	Limited, sometimes supported	Built-in metadata via document fields
Typical Access Pattern	`get(key)`	`find(where field = value)`
Performance Use Case	Extremely fast lookups	Flexible querying with good performance
Examples	Redis, Riak, DynamoDB	MongoDB, CouchDB
Best Use Case	Simple, high-speed data access	Semi-structured, query-heavy data
Query Tools	External tools (e.g., Solr integration)	Native query language

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 Cassandra.

These databases with a bigtable-style data model are often referred to as column stores, but that name has been around for a while to describe a different animal. Pre-NoSQL column stores, such as C-Store [C-Store], were happy with SQL and the relational model. The thing that made them different was the way in which they physically stored data. Most databases have a row as a unit of storage which, in particular, helps write performance. However, there are many scenarios where writes are

rare, but you often need to read a few columns of many rows at once. In this situation, it's better to store groups of columns for all rows as the basic storage unit—which is why these databases are called column stores.

Bigtable and its offspring follow this notion of storing groups of columns (column families) together, but part company with C-Store and friends by abandoning the relational model and SQL. In this book, we refer to this class of databases as column-family databases.

Perhaps the best way to think of the column-family model is as a two-level aggregate structure. As with key-value stores, the first key is often described as a row identifier, picking up the aggregate of interest. The difference with column-family structures is that this row aggregate is itself formed of a map of more detailed values. These second-level values are referred to as columns. As well as accessing the row as a whole, operations also allow picking out a particular column, so to get a particular customer's name from Figure 2.5 you could do something like `get('1234', 'name')`.

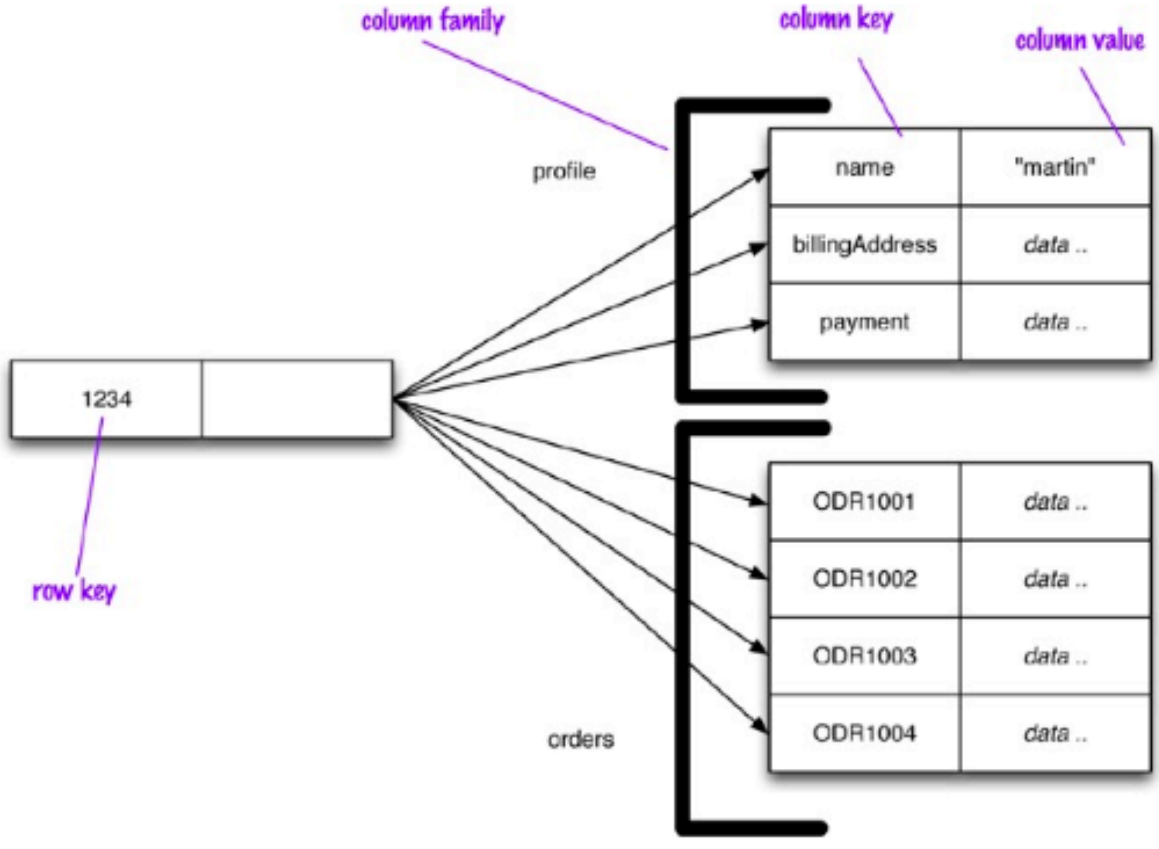

Figure 2.5. Representing customer information in a column-family structure

Column-family databases organize their columns into column families. Each column has to be part of a single column family, and the column acts as unit for access, with the assumption that data for a particular column family will be usually accessed together.

This also gives you a couple of ways to think about how the data is structured.

- Row-oriented: Each row is an aggregate (for example, customer with the ID of 1234) with column families representing useful chunks of data (profile, order history) within that aggregate.
- Column-oriented: Each column family defines a record type (e.g., customer profiles) with rows for each of the records. You then think of a row as the join of records in all column families.

This latter aspect reflects the columnar nature of column-family databases. Since the database knows about these common groupings of data, it can use this information for its storage and access behavior. Even though a document database declares some structure to the database, each document is

still seen as a single unit. Column families give a two-dimensional quality to column-family databases.

This terminology is as established by Google Bigtable and HBase, but Cassandra looks at things slightly differently. A row in Cassandra only occurs in one column family, but that column family may contain supercolumns—columns that contain nested columns. The supercolumns in Cassandra are the best equivalent to the classic Bigtable column families.

It can still be confusing to think of column-families as tables. You can add any column to any row, and rows can have very different column keys. While new columns are added to rows during regular database access, defining new column families is much rarer and may involve stopping the database for it to happen.

The example of Figure 2.5 illustrates another aspect of column-family databases that may be unfamiliar for people used to relational tables: the `orders` column family. Since columns can be added freely, you can model a list of items by making each item a separate column. This is very odd if you think of a column family as a table, but quite natural if you think of a column-family row as an aggregate. Cassandra uses the terms "wide" and "skinny." **Skinny rows** have few columns with the same columns used across the many different rows. In this case, the column family defines a record type, each row is a record, and each column is a field. A **wide row** has many columns (perhaps thousands), with rows having very different columns. A wide column family models a list, with each column being one element in that list.

A consequence of wide column families is that a column family may define a sort order for its columns. This way we can access orders by their order key and access ranges of orders by their keys. While this might not be useful if we keyed orders by their IDs, it would be if we made the key out of a concatenation of date and ID (e.g., `20111027-1001`).

Although it's useful to distinguish column families by their wide or skinny nature, there's no technical reason why a column family cannot contain both field-like columns and list-like columns—although doing this would confuse the sort ordering.

DISTRIBUTION MODELS:

→ the primary driver of interest in NoSQL has been its ability to run databases on a large cluster.

→ 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.

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.

⇒ Sharding:

→often, a busy datastore 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.

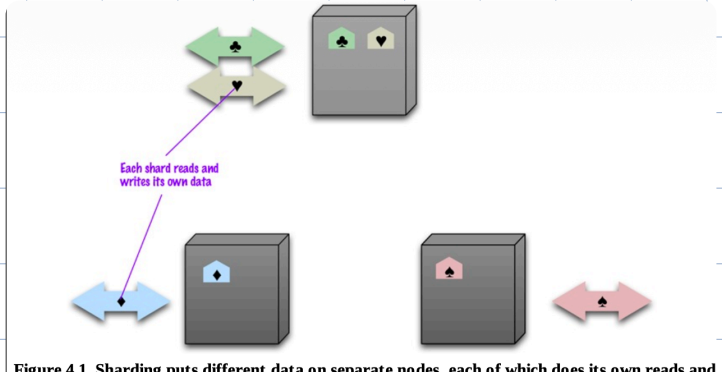

Figure 4.1. Sharding puts different data on separate nodes, each of which does its own reads and writes.

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.

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).

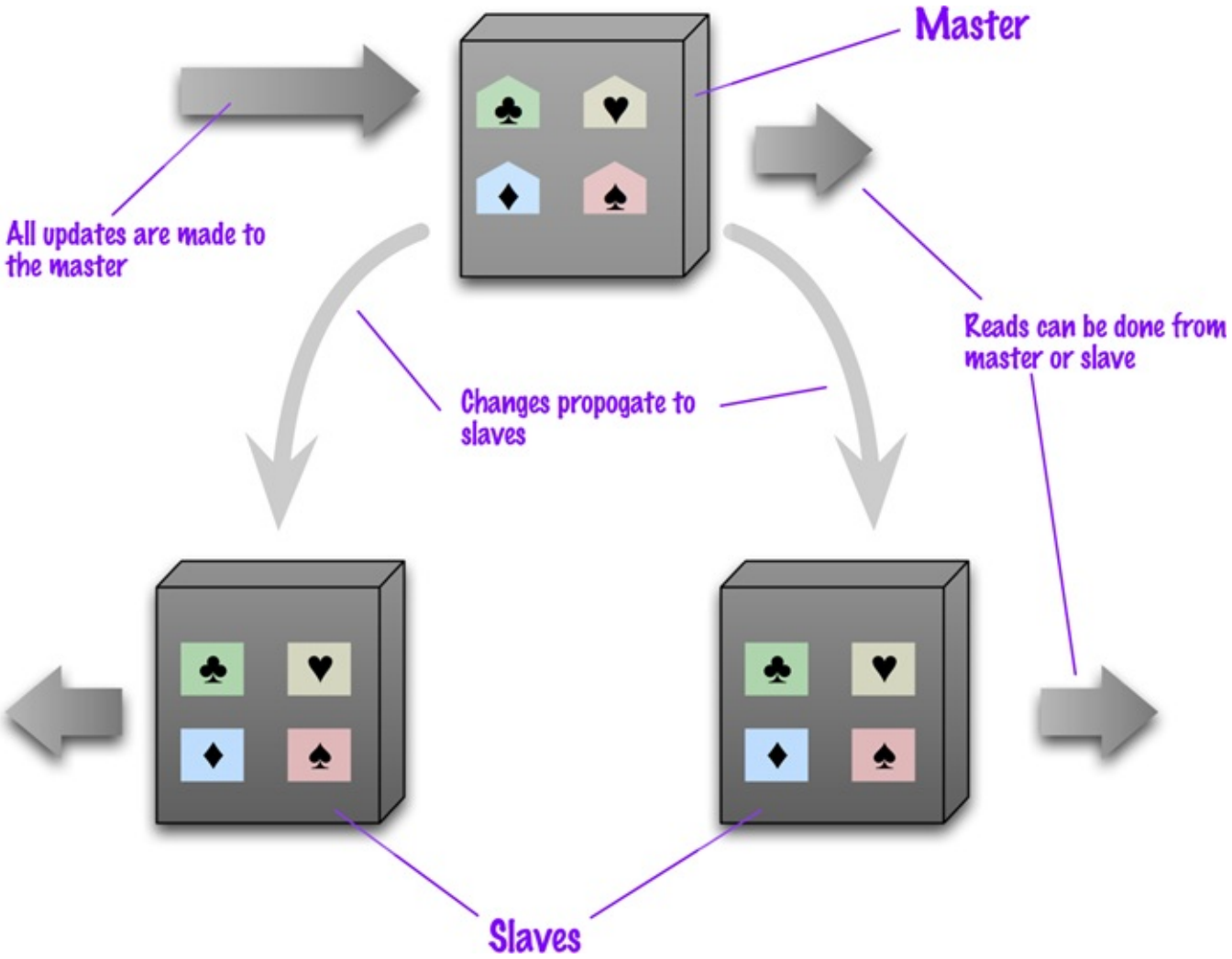

Figure 4.2. Data is replicated from master to slaves. The master services all writes; reads may come from either master or slaves.

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.

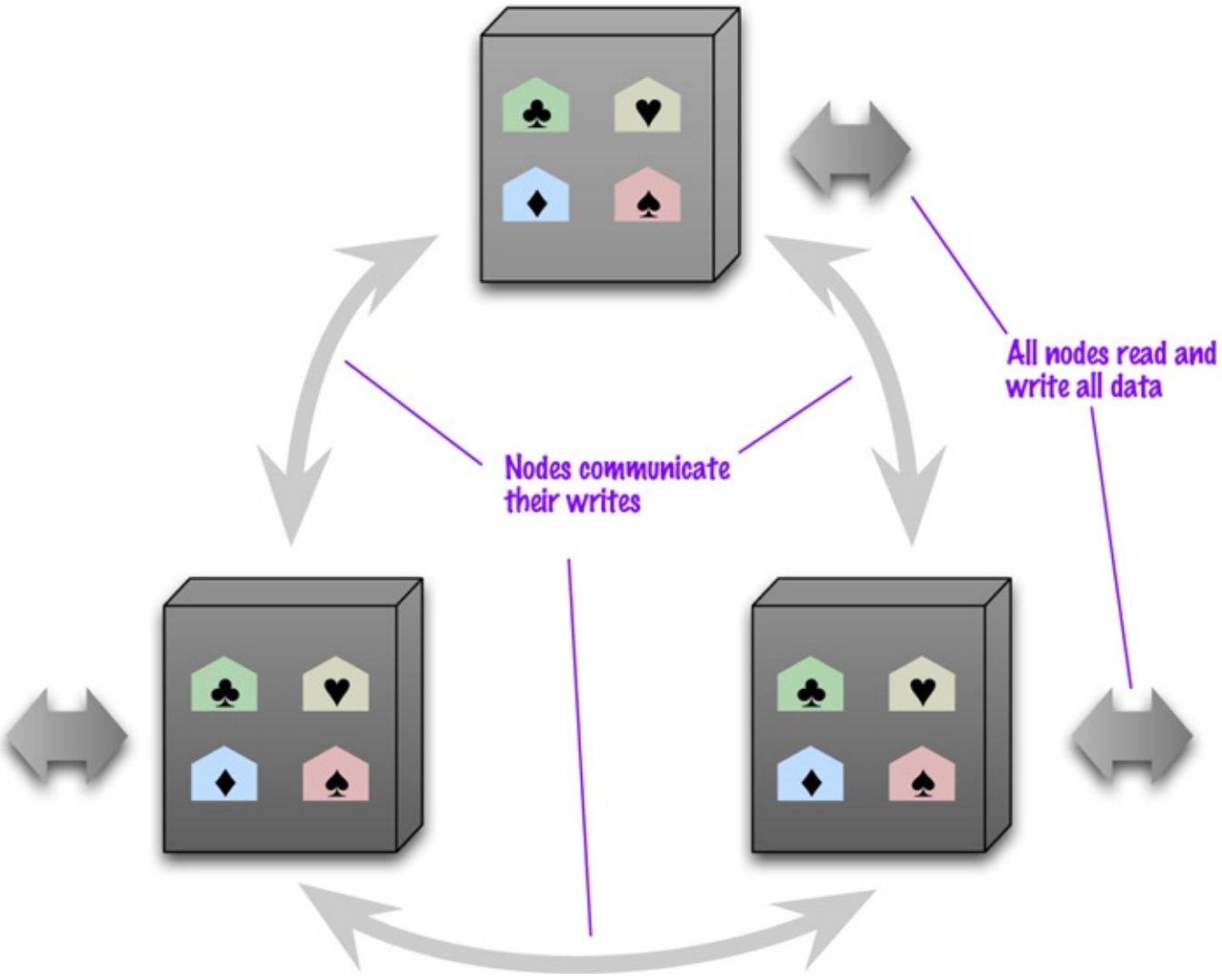

Figure 4.3. Peer-to-peer replication has all nodes applying reads and writes to all the data.

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.

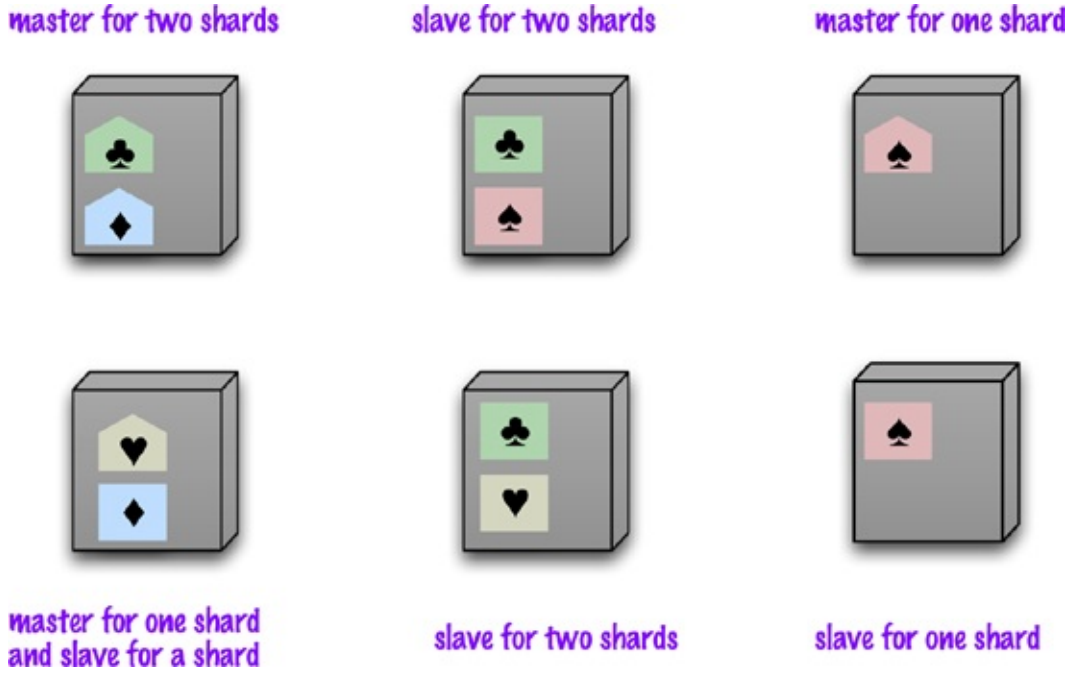

Figure 4.4. Using master-slave replication together with sharding

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).

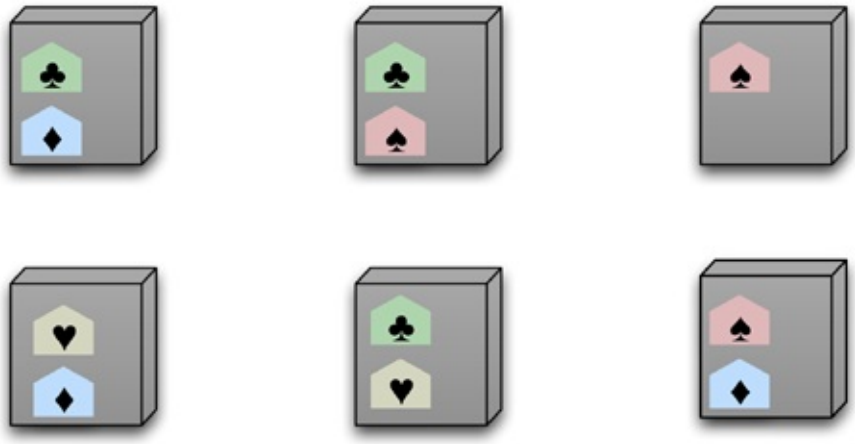

Figure 4.5. Using peer-to-peer replication together with sharding

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.