Part I: Understand

Chapter 1. Why NoSQL?

For almost as long as we've been in the software profession, relational databases have been the default choice for serious data storage, especially in the world of enterprise applications. If you're an architect starting a new project, your only choice is likely to be which relational database to use. (And often not even that, if your company has a dominant vendor.) There have been times when a database technology threatened to take a piece of the action, such as object databases in the 1990's, but these alternatives never got anywhere.

After such a long period of dominance, the current excitement about NoSQL databases comes as a surprise. In this chapter we'll explore why relational databases became so dominant, and why we think the current rise of NoSQL databases isn't a flash in the pan.

1.1. The Value of Relational Databases

Relational databases have become such an embedded part of our computing culture that it's easy to take them for granted. It's therefore useful to revisit the benefits they provide.

1.1.1. Getting at Persistent Data

Probably the most obvious value of a database is keeping large amounts of persistent data. Most computer architectures have the notion of two areas of memory: a fast volatile "main memory" and a larger but slower "backing store." Main memory is both limited in space and loses all data when you lose power or something bad happens to the operating system. Therefore, to keep data around, we write it to a backing store, commonly seen a disk (although these days that disk can be persistent memory).

The backing store can be organized in all sorts of ways. For many productivity applications (such as word processors), it's a file in the file system of the operating system. For most enterprise applications, however, the backing store is a database. The database allows more flexibility than a file system in storing large amounts of data in a way that allows an application program to get at small bits of that information quickly and easily.

1.1.2. Concurrency

Enterprise applications tend to have many people looking at the same body of data at once, possibly modifying that data. Most of the time they are working on different areas of that data, but occasionally they operate on the same bit of data. As a result, we have to worry about coordinating these interactions to avoid such things as double booking of hotel rooms.

Concurrency is notoriously difficult to get right, with all sorts of errors that can trap even the most careful programmers. Since enterprise applications can have lots of users and other systems all working concurrently, there's a lot of room for bad things to happen. Relational databases help handle this by controlling all access to their data

through transactions. While this isn't a cure-all (you still have to handle a transactional error when you try to book a room that's just gone), the transactional mechanism has worked well to contain the complexity of concurrency.

Transactions also play a role in error handling. With transactions, you can make a change, and if an error occurs during the processing of the change you can roll back the transaction to clean things up.

1.1.3. Integration

Enterprise applications live in a rich ecosystem that requires multiple applications, written by different teams, to collaborate in order to get things done. This kind of inter-application collaboration is awkward because it means pushing the human organizational boundaries. Applications often need to use the same data and updates made through one application have to be visible to others.

A common way to do this is **shared database integration** [Hohpe and Woolf] where multiple applications store their data in a single database. Using a single database allows all the applications to use each others' data easily, while the database's concurrency control handles multiple applications in the same way as it handles multiple users in a single application.

1.1.4. A (Mostly) Standard Model

Relational databases have succeeded because they provide the core benefits we outlined earlier in a (mostly) standard way. As a result, developers and database professionals can learn the basic relational model and apply it in many projects. Although there are differences between different relational databases, the core mechanisms remain the same: Different vendors' SQL dialects are similar, transactions operate in mostly the same way.

1.2. Impedance Mismatch

Relational databases provide many advantages, but they are by no means perfect. Even from their early days, there have been lots of frustrations with them.

For application developers, the biggest frustration has been what's commonly called the **impedance mismatch**: the difference between the relational model and the in-memory data structures. The relational data model organizes data into a structure of tables and rows, or more properly, relations and tuples. In the relational model, a **tuple** is a set of name-value pairs and a **relation** is a set of tuples. (The relational definition of a tuple is slightly different from that in mathematics and many programming languages with a tuple data type, where a tuple is a sequence of values.) All operations in SQL consume and return relations, which leads to the mathematically elegant relational algebra.

This foundation on relations provides a certain elegance and simplicity, but it also introduces limitations. In particular, the values in a relational tuple have to be simple—they cannot contain any structure, such as a nested record or a list. This limitation isn't true for in-memory data structures, which can take on much richer structures than relations. As a result, if you want to use a richer in-memory data structure, you have to

translate it to a relational representation to store it on disk. Hence the impedance mismatch—two different representations that require translation (see Figure 1.1).

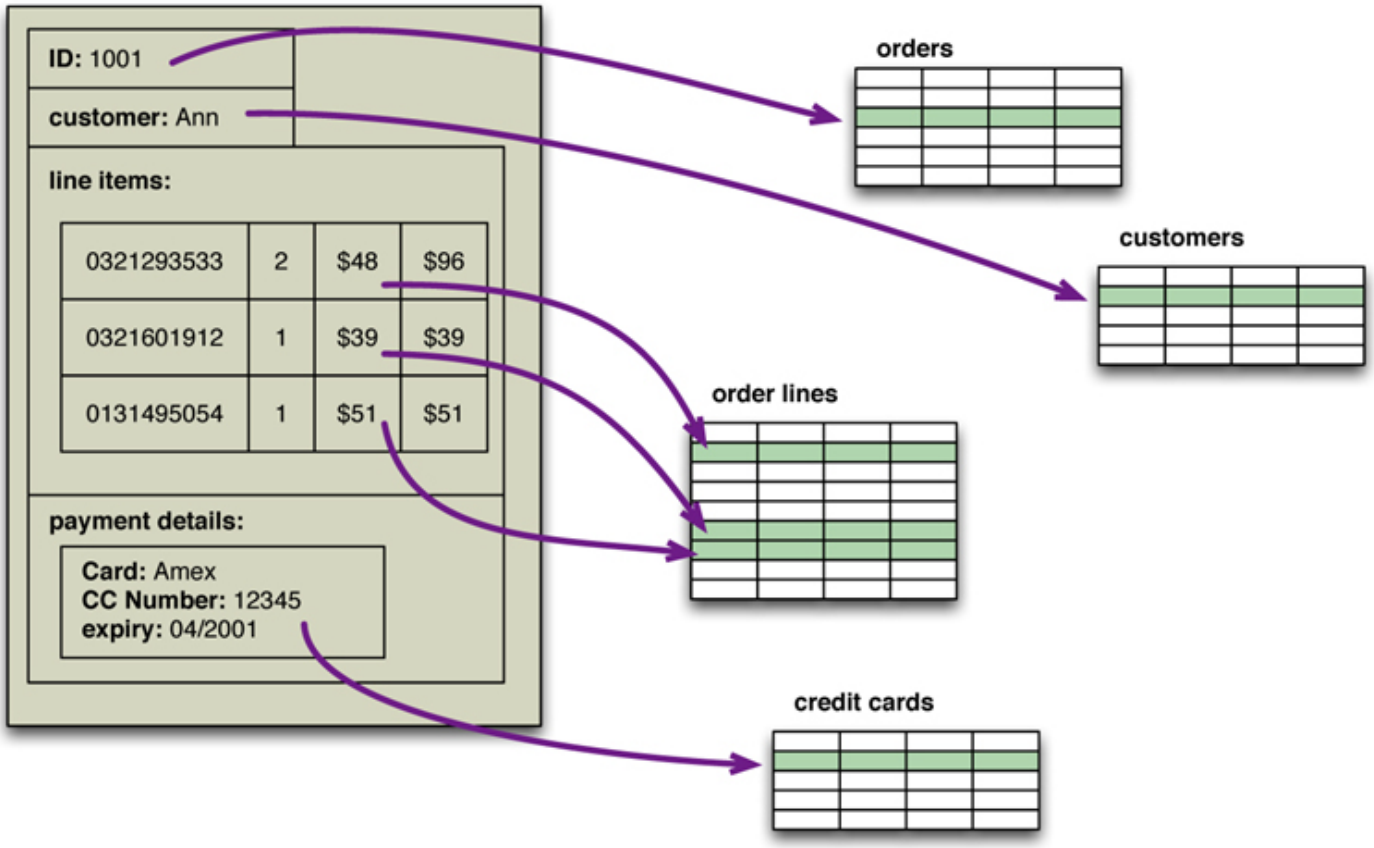

Figure 1.1. An order, which looks like a single aggregate structure in the UI, is split into many rows from many tables in a relational database

The impedance mismatch is a major source of frustration to application developers, and in the 1990s many people believed that it would lead to relational databases being replaced with databases that replicate the in-memory data structures to disk. That decade was marked with the growth of object-oriented programming languages, and with them came object-oriented databases—both looking to be the dominant environment for software development in the new millennium.

However, while object-oriented languages succeeded in becoming the major force in programming, object-oriented databases faded into obscurity. Relational databases saw off the challenge by stressing their role as an integration mechanism, supported by a mostly standard language of data manipulation (SQL) and a growing professional divide between application developers and database administrators.

Impedance mismatch has been made much easier to deal with by the wide availability of object-relational mapping frameworks, such as Hibernate and iBATIS that implement well-known mapping patterns [Fowler PoEAA], but the mapping problem is still an issue. Object-relational mapping frameworks remove a lot of grunt work, but can become a problem of their own when people try too hard to ignore the database and query performance suffers.

Relational databases continued to dominate the enterprise computing world in the 2000s, but during that decade cracks began to open in their dominance.

1.3. Application and Integration Databases

The exact reasons why relational databases triumphed over OO databases are still the subject of an occasional pub debate for developers of a certain age. But in our view, the primary factor was the role of SQL as an integration mechanism between applications. In this scenario, the database acts as an **integration database**—with multiple applications, usually developed by separate teams, storing their data in a common database. This improves communication because all the applications are operating on a consistent set of persistent data.

There are downsides to shared database integration. A structure that's designed to integrate many applications ends up being more complex—indeed, often dramatically more complex—than any single application needs. Furthermore, should an application want to make changes to its data storage, it needs to coordinate with all the other applications using the database. Different applications have different structural and performance needs, so an index required by one application may cause a problematic hit on inserts for another. The fact that each application is usually a separate team also means that the database usually cannot trust applications to update the data in a way that preserves database integrity and thus needs to take responsibility for that within the database itself.

A different approach is to treat your database as an **application database**—which is only directly accessed by a single application codebase that's looked after by a single team. With an application database, only the team using the application needs to know about the database structure, which makes it much easier to maintain and evolve the schema. Since the application team controls both the database and the application code, the responsibility for database integrity can be put in the application code.

Interoperability concerns can now shift to the interfaces of the application, allowing for better interaction protocols and providing support for changing them. During the 2000s we saw a distinct shift to web services [Daigneau], where applications would communicate over HTTP. Web services enabled a new form of a widely used communication mechanism—a challenger to using the SQL with shared databases. (Much of this work was done under the banner of "Service-Oriented Architecture"—a term most notable for its lack of a consistent meaning.)

An interesting aspect of this shift to web services as an integration mechanism was that it resulted in more flexibility for the structure of the data that was being exchanged. If you communicate with SQL, the data must be structured as relations. However, with a service, you are able to use richer data structures with nested records and lists. These are usually represented as documents in XML or, more recently, JSON. In general, with remote communication you want to reduce the number of round trips involved in the interaction, so it's useful to be able to put a rich structure of information into a single request or response.

If you are going to use services for integration, most of the time web services—using text over HTTP—is the way to go. However, if you are dealing with highly performance-sensitive interactions, you may need a binary protocol. Only do this if you are sure you have the need, as text protocols are easier to work with—consider the example of the Internet.

Once you have made the decision to use an application database, you get more freedom of choosing a database. Since there is a decoupling between your internal database and the services with which you talk to the outside world, the outside world doesn't have to care how you store your data, allowing you to consider nonrelational options. Furthermore, there are many features of relational databases, such as security, that are less useful to an application database because they can be done by the enclosing application instead.

Despite this freedom, however, it wasn't apparent that application databases led to a big rush to alternative data stores. Most teams that embraced the application database approach stuck with relational databases. After all, using an application database yields many advantages even ignoring the database flexibility (which is why we generally recommend it). Relational databases are familiar and usually work very well or, at least, well enough. Perhaps, given time, we might have seen the shift to application databases to open a real crack in the relational hegemony—but such cracks came from another source.

1.4. Attack of the Clusters

At the beginning of the new millennium the technology world was hit by the busting of the 1990s dot-com bubble. While this saw many people questioning the economic future of the Internet, the 2000s did see several large web properties dramatically increase in scale.

This increase in scale was happening along many dimensions. Websites started tracking activity and structure in a very detailed way. Large sets of data appeared: links, social networks, activity in logs, mapping data. With this growth in data came a growth in users—as the biggest websites grew to be vast estates regularly serving huge numbers of visitors.

Coping with the increase in data and traffic required more computing resources. To handle this kind of increase, you have two choices: up or out. Scaling up implies bigger machines, more processors, disk storage, and memory. But bigger machines get more and more expensive, not to mention that there are real limits as your size increases. The alternative is to use lots of small machines in a cluster. A cluster of small machines can use commodity hardware and ends up being cheaper at these kinds of scales. It can also be more resilient—while individual machine failures are common, the overall cluster can be built to keep going despite such failures, providing high reliability.

As large properties moved towards clusters, that revealed a new problem—relational databases are not designed to be run on clusters. Clustered relational databases, such as the Oracle RAC or Microsoft SQL Server, work on the concept of a shared disk subsystem. They use a cluster-aware file system that writes to a highly available disk subsystem—but this means the cluster still has the disk subsystem as a single point of failure. Relational databases could also be run as separate servers for different sets of data, effectively sharding ("Sharding," p. 38) the database. While this separates the load, all the sharding has to be controlled by the application which has to keep track of which database server to talk to for each bit of data. Also, we lose any querying,

referential integrity, transactions, or consistency controls that cross shards. A phrase we often hear in this context from people who've done this is "unnatural acts."

These technical issues are exacerbated by licensing costs. Commercial relational databases are usually priced on a single-server assumption, so running on a cluster raised prices and led to frustrating negotiations with purchasing departments.

This mismatch between relational databases and clusters led some organization to consider an alternative route to data storage. Two companies in particular—Google and Amazon—have been very influential. Both were on the forefront of running large clusters of this kind; furthermore, they were capturing huge amounts of data. These things gave them the motive. Both were successful and growing companies with strong technical components, which gave them the means and opportunity. It was no wonder they had murder in mind for their relational databases. As the 2000s drew on, both companies produced brief but highly influential papers about their efforts: BigTable from Google and Dynamo from Amazon.

It's often said that Amazon and Google operate at scales far removed from most organizations, so the solutions they needed may not be relevant to an average organization. While it's true that most software projects don't need that level of scale, it's also true that more and more organizations are beginning to explore what they can do by capturing and processing more data—and to run into the same problems. So, as more information leaked out about what Google and Amazon had done, people began to explore making databases along similar lines—explicitly designed to live in a world of clusters. While the earlier menaces to relational dominance turned out to be phantoms, the threat from clusters was serious.

1.5. The Emergence of NoSQL

It's a wonderful irony that the term "NoSQL" first made its appearance in the late 90s as the name of an open-source relational database [Strozzi NoSQL]. Led by Carlo Strozzi, this database stores its tables as ASCII files, each tuple represented by a line with fields separated by tabs. The name comes from the fact that the database doesn't use SQL as a query language. Instead, the database is manipulated through shell scripts that can be combined into the usual UNIX pipelines. Other than the terminological coincidence, Strozzi's NoSQL had no influence on the databases we describe in this book.

The usage of "NoSQL" that we recognize today traces back to a meetup on June 11, 2009 in San Francisco organized by Johan Oskarsson, a software developer based in London. The example of BigTable and Dynamo had inspired a bunch of projects experimenting with alternative data storage, and discussions of these had become a feature of the better software conferences around that time. Johan was interested in finding out more about some of these new databases while he was in San Francisco for a Hadoop summit. Since he had little time there, he felt that it wouldn't be feasible to visit them all, so he decided to host a meetup where they could all come together and present their work to whoever was interested.

Johan wanted a name for the meetup—something that would make a good Twitter hashtag: short, memorable, and without too many Google hits so that a search on the name would quickly find the meetup. He asked for suggestions on the #cassandra IRC channel and got a few, selecting the suggestion of “NoSQL” from Eric Evans (a developer at Rackspace, no connection to the DDD Eric Evans). While it had the disadvantage of being negative and not really describing these systems, it did fit the hashtag criteria. At the time they were thinking of only naming a single meeting and were not expecting it to catch on to name this entire technology trend [Oskarsson].

The term “NoSQL” caught on like wildfire, but it’s never been a term that’s had much in the way of a strong definition. The original call [NoSQL Meetup] for the meetup asked for “open-source, distributed, nonrelational databases.” The talks there [NoSQL Debrief] were from Voldemort, Cassandra, Dynomite, HBase, Hypertable, CouchDB, and MongoDB—but the term has never been confined to that original septet. There’s no generally accepted definition, nor an authority to provide one, so all we can do is discuss some common characteristics of the databases that tend to be called “NoSQL.”

To begin with, there is the obvious point that NoSQL databases don’t use SQL. Some of them do have query languages, and it makes sense for them to be similar to SQL in order to make them easier to learn. Cassandra’s CQL is like this—“exactly like SQL (except where it’s not)” [CQL]. But so far none have implemented anything that would fit even the rather flexible notion of standard SQL. It will be interesting to see what happens if an established NoSQL database decides to implement a reasonably standard SQL; the only predictable outcome for such an eventuality is plenty of argument.

Another important characteristic of these databases is that they are generally open-source projects. Although the term NoSQL is frequently applied to closed-source systems, there’s a notion that NoSQL is an open-source phenomenon.

Most NoSQL databases are driven by the need to run on clusters, and this is certainly true of those that were talked about during the initial meetup. This has an effect on their data model as well as their approach to consistency. Relational databases use ACID transactions (p. 19) to handle consistency across the whole database. This inherently clashes with a cluster environment, so NoSQL databases offer a range of options for consistency and distribution.

However, not all NoSQL databases are strongly oriented towards running on clusters. Graph databases are one style of NoSQL databases that uses a distribution model similar to relational databases but offers a different data model that makes it better at handling data with complex relationships.

NoSQL databases are generally based on the needs of the early 21st century web estates, so usually only systems developed during that time frame are called NoSQL—thus ruling out hoards of databases created before the new millennium, let alone BC (Before Codd).

NoSQL databases operate without a schema, allowing you to freely add fields to database records without having to define any changes in structure first. This is particularly useful when dealing with nonuniform data and custom fields which forced

relational databases to use names like `customField6` or custom field tables that are awkward to process and understand.

All of the above are common characteristics of things that we see described as NoSQL databases. None of these are definitional, and indeed it's likely that there will never be a coherent definition of "NoSQL" (sigh). However, this crude set of characteristics has been our guide in writing this book. Our chief enthusiasm with this subject is that the rise of NoSQL has opened up the range of options for data storage. Consequently, this opening up shouldn't be confined to what's usually classed as a NoSQL store. We hope that other data storage options will become more acceptable, including many that predate the NoSQL movement. There is a limit, however, to what we can usefully discuss in this book, so we've decided to concentrate on this noDefinition.

When you first hear "NoSQL," an immediate question is what does it stand for—a "no" to SQL? Most people who talk about NoSQL say that it really means "Not Only SQL," but this interpretation has a couple of problems. Most people write "NoSQL" whereas "Not Only SQL" would be written "NOSQL." Also, there wouldn't be much point in calling something a NoSQL database under the "not only" meaning—because then, Oracle or Postgres would fit that definition, we would prove that black equals white and would all get run over on crosswalks.

To resolve this, we suggest that you don't worry about what the term stands for, but rather about what it means (which is recommended with most acronyms). Thus, when "NoSQL" is applied to a database, it refers to an ill-defined set of mostly open-source databases, mostly developed in the early 21st century, and mostly not using SQL.

The "not-only" interpretation does have its value, as it describes the ecosystem that many people think is the future of databases. This is in fact what we consider to be the most important contribution of this way of thinking—it's better to think of NoSQL as a movement rather than a technology. We don't think that relational databases are going away—they are still going to be the most common form of database in use. Even though we've written this book, we still recommend relational databases. Their familiarity, stability, feature set, and available support are compelling arguments for most projects.

The change is that now we see relational databases as one option for data storage. This point of view is often referred to as **polyglot persistence**—using different data stores in different circumstances. Instead of just picking a relational database because everyone does, we need to understand the nature of the data we're storing and how we want to manipulate it. The result is that most organizations will have a mix of data storage technologies for different circumstances.

In order to make this polyglot world work, our view is that organizations also need to shift from integration databases to application databases. Indeed, we assume in this book that you'll be using a NoSQL database as an application database; we don't generally consider NoSQL databases a good choice for integration databases. We don't see this as a disadvantage as we think that even if you don't use NoSQL, shifting to encapsulating data in services is a good direction to take.

In our account of the history of NoSQL development, we've concentrated on big data running on clusters. While we think this is the key thing that drove the opening up of the database world, it isn't the only reason we see project teams considering NoSQL databases. An equally important reason is the old frustration with the impedance mismatch problem. The big data concerns have created an opportunity for people to think freshly about their data storage needs, and some development teams see that using a NoSQL database can help their productivity by simplifying their database access even if they have no need to scale beyond a single machine.

So, as you read the rest of this book, remember there are two primary reasons for considering NoSQL. One is to handle data access with sizes and performance that demand a cluster; the other is to improve the productivity of application development by using a more convenient data interaction 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.

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.

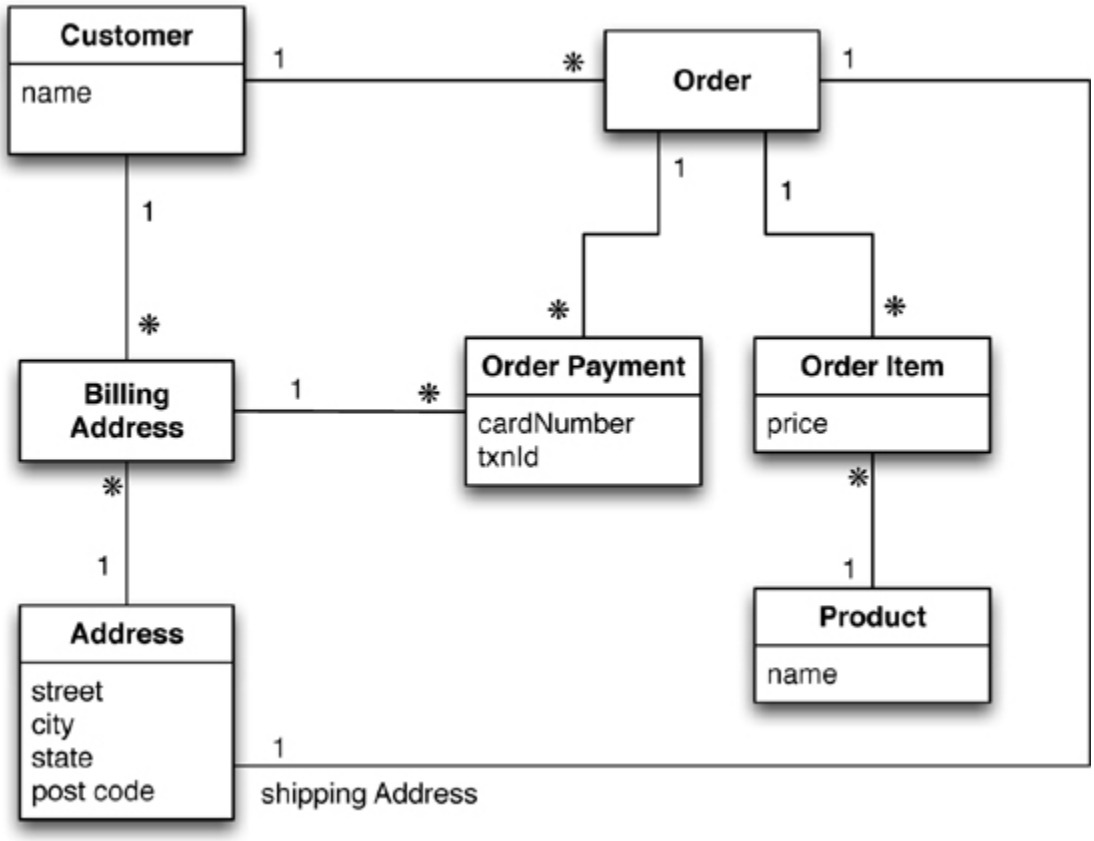

Figure 2.1. Data model oriented around a relational database (using UML notation [Fowler UML])

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).

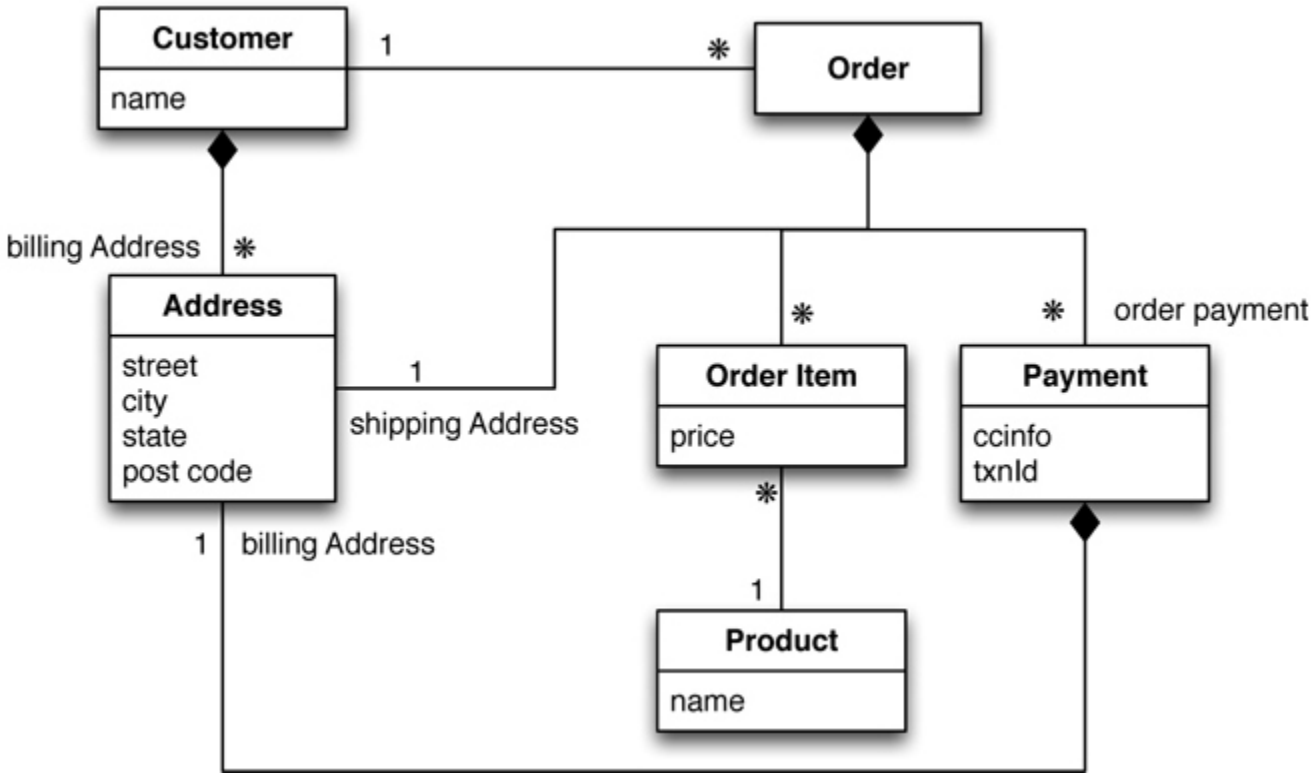

Figure 2.3. An aggregate data model

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).

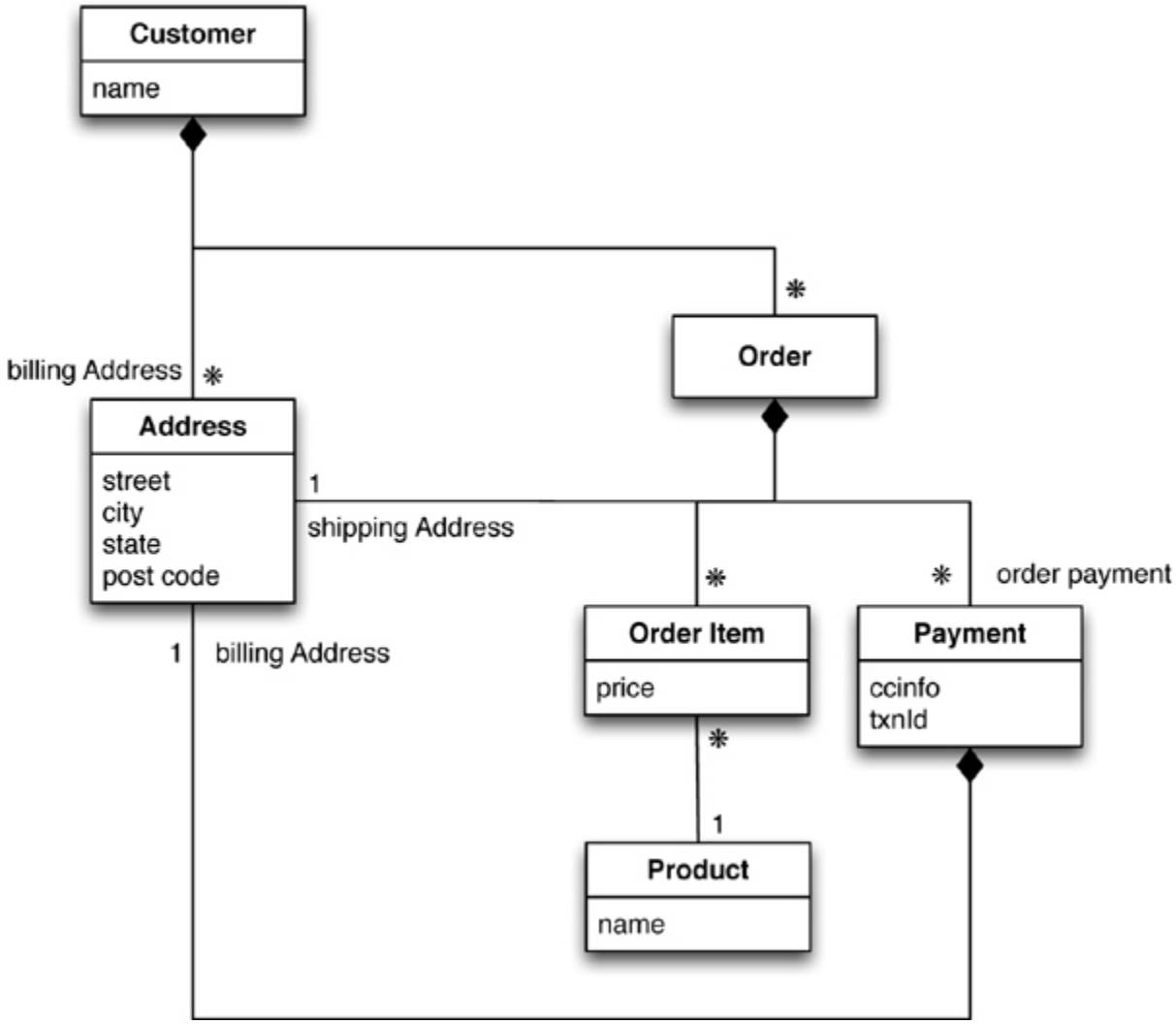

Figure 2.4. Embed all the objects for customer and the customer's orders

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 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')`.

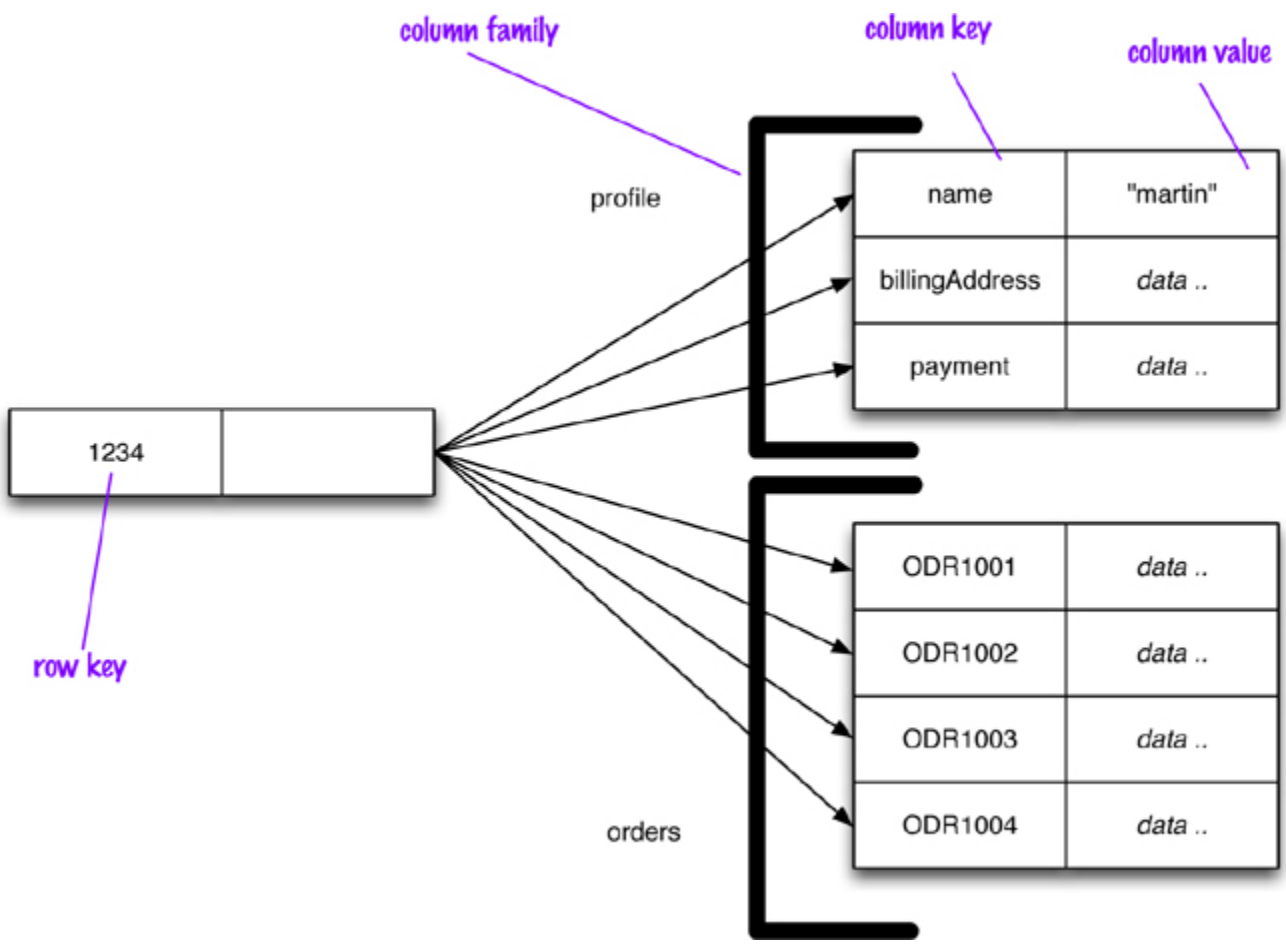

Figure 2.5. Representing customer information in a column-family structure

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.

2.4. Summarizing Aggregate-Oriented Databases

At this point, we've covered enough material to give you a reasonable overview of the three different styles of aggregate-oriented data models and how they differ.

What they all share is the notion of an aggregate indexed by a key that you can use for lookup. This aggregate is central to running on a cluster, as the database will ensure that all the data for an aggregate is stored together on one node. The aggregate also acts as the atomic unit for updates, providing a useful, if limited, amount of transactional control.

Within that notion of aggregate, we have some differences. The key-value data model treats the aggregate as an opaque whole, which means you can only do key lookup for the whole aggregate—you cannot run a query nor retrieve a part of the aggregate.

The document model makes the aggregate transparent to the database allowing you to do queries and partial retrievals. However, since the document has no schema, the database cannot act much on the structure of the document to optimize the storage and retrieval of parts of the aggregate.

Column-family models divide the aggregate into column families, allowing the database to treat them as units of data within the row aggregate. This imposes some structure on the aggregate but allows the database to take advantage of that structure to improve its accessibility.

2.5. Further Reading

For more on the general concept of aggregates, which are often used with relational databases too, see [Evans]. The Domain-Driven Design community is the best source for further information about aggregates—recent information usually appears at http://domaindrivendesign.org.

2.6. Key Points

- An aggregate is a collection of data that we interact with as a unit. Aggregates form the boundaries for ACID operations with the database.
- Key-value, document, and column-family databases can all be seen as forms of aggregate-oriented database.
- Aggregates make it easier for the database to manage data storage over clusters.
- Aggregate-oriented databases work best when most data interaction is done with the same aggregate; aggregate-ignorant databases are better when interactions use data organized in many different formations.

Chapter 3. More Details on Data Models

So far we've covered the key feature in most NoSQL databases: their use of aggregates and how aggregate-oriented databases model aggregates in different ways. While aggregates are a central part of the NoSQL story, there is more to the data modeling side than that, and we'll explore these further concepts in this chapter.

3.1. Relationships

Aggregates are useful in that they put together data that is commonly accessed together. But there are still lots of cases where data that's related is accessed differently. Consider the relationship between a customer and all of his orders. Some applications will want to access the order history whenever they access the customer; this fits in well with combining the customer with his order history into a single aggregate. Other applications, however, want to process orders individually and thus model orders as independent aggregates.

In this case, you'll want separate order and customer aggregates but with some kind of relationship between them so that any work on an order can look up customer data. The simplest way to provide such a link is to embed the ID of the customer within the order's aggregate data. That way, if you need data from the customer record, you read the order, ferret out the customer ID, and make another call to the database to read the customer data. This will work, and will be just fine in many scenarios—but the database will be ignorant of the relationship in the data. This can be important because there are times when it's useful for the database to know about these links.

As a result, many databases—even key-value stores—provide ways to make these relationships visible to the database. Document stores make the content of the aggregate available to the database to form indexes and queries. Riak, a key-value store, allows you to put link information in metadata, supporting partial retrieval and link-walking capability.

An important aspect of relationships between aggregates is how they handle updates. Aggregate-oriented databases treat the aggregate as the unit of data-retrieval. Consequently, atomicity is only supported within the contents of a single aggregate. If you update multiple aggregates at once, you have to deal yourself with a failure partway through. Relational databases help you with this by allowing you to modify multiple records in a single transaction, providing ACID guarantees while altering many rows.

All of this means that aggregate-oriented databases become more awkward as you need to operate across multiple aggregates. There are various ways to deal with this, which we'll explore later in this chapter, but the fundamental awkwardness remains.

This may imply that if you have data based on lots of relationships, you should prefer a relational database over a NoSQL store. While that's true for aggregate-oriented databases, it's worth remembering that relational databases aren't all that stellar with complex relationships either. While you can express queries involving joins in SQL,

things quickly get very hairy—both with SQL writing and with the resulting performance—as the number of joins mounts up.

This makes it a good moment to introduce another category of databases that's often lumped into the NoSQL pile.

3.2. Graph Databases

Graph databases are an odd fish in the NoSQL pond. Most NoSQL databases were inspired by the need to run on clusters, which led to aggregate-oriented data models of large records with simple connections. Graph databases are motivated by a different frustration with relational databases and thus have an opposite model—small records with complex interconnections, something like Figure 3.1.

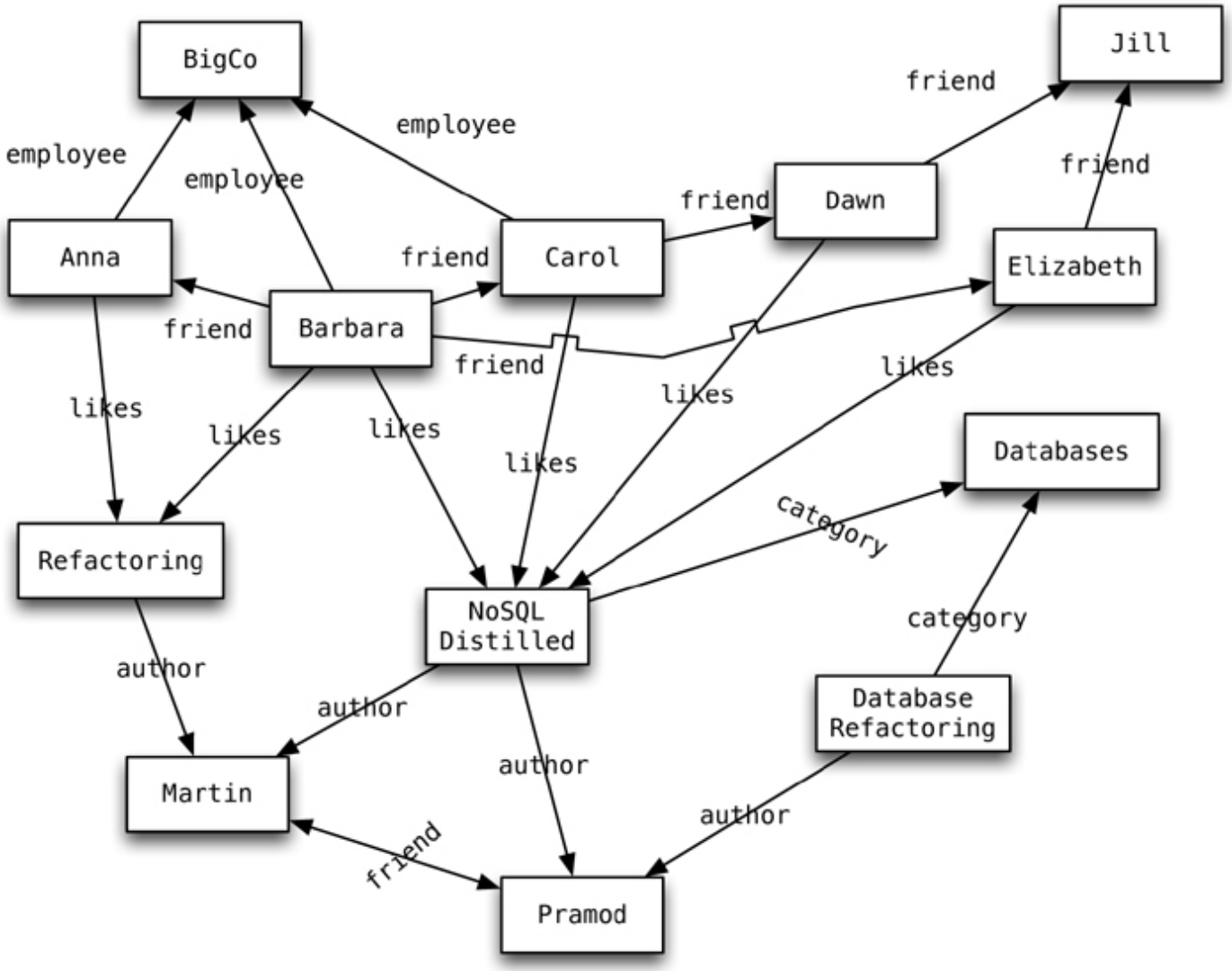

Figure 3.1. An example graph structure

In this context, a graph isn't a bar chart or histogram; instead, we refer to a graph data structure of nodes connected by edges.

In Figure 3.1 we have a web of information whose nodes are very small (nothing more than a name) but there is a rich structure of interconnections between them. With this structure, we can ask questions such as "find the books in the Databases category that are written by someone whom a friend of mine likes."

Graph databases specialize in capturing this sort of information—but on a much larger scale than a readable diagram could capture. This is ideal for capturing any data consisting of complex relationships such as social networks, product preferences, or eligibility rules.

The fundamental data model of a graph database is very simple: nodes connected by edges (also called arcs). Beyond this essential characteristic there is a lot of variation in data models—in particular, what mechanisms you have to store data in your nodes and edges. A quick sample of some current capabilities illustrates this variety of possibilities: FlockDB is simply nodes and edges with no mechanism for additional attributes; Neo4J allows you to attach Java objects as properties to nodes and edges in a schemaless fashion (“Features,” p. 113); Infinite Graph stores your Java objects, which are subclasses of its built-in types, as nodes and edges.

Once you have built up a graph of nodes and edges, a graph database allows you to query that network with query operations designed with this kind of graph in mind. This is where the important differences between graph and relational databases come in. Although relational databases can implement relationships using foreign keys, the joins required to navigate around can get quite expensive—which means performance is often poor for highly connected data models. Graph databases make traversal along the relationships very cheap. A large part of this is because graph databases shift most of the work of navigating relationships from query time to insert time. This naturally pays off for situations where querying performance is more important than insert speed.

Most of the time you find data by navigating through the network of edges, with queries such as “tell me all the things that both Anna and Barbara like.” You do need a starting place, however, so usually some nodes can be indexed by an attribute such as ID. So you might start with an ID lookup (i.e., look up the people named “Anna” and “Barbara”) and then start using the edges. Still, graph databases expect most of your query work to be navigating relationships.

The emphasis on relationships makes graph databases very different from aggregate-oriented databases. This data model difference has consequences in other aspects, too; you’ll find such databases are more likely to run on a single server rather than distributed across clusters. ACID transactions need to cover multiple nodes and edges to maintain consistency. The only thing they have in common with aggregate-oriented databases is their rejection of the relational model and an upsurge in attention they received around the same time as the rest of the NoSQL field.

3.3. Schemaless Databases

A common theme across all the forms of NoSQL databases is that they are schemaless. When you want to store data in a relational database, you first have to define a schema—a defined structure for the database which says what tables exist, which columns exist, and what data types each column can hold. Before you store some data, you have to have the schema defined for it.

With NoSQL databases, storing data is much more casual. A key-value store allows you to store any data you like under a key. A document database effectively does the

same thing, since it makes no restrictions on the structure of the documents you store. Column-family databases allow you to store any data under any column you like. Graph databases allow you to freely add new edges and freely add properties to nodes and edges as you wish.

Advocates of schemalessness rejoice in this freedom and flexibility. With a schema, you have to figure out in advance what you need to store, but that can be hard to do. Without a schema binding you, you can easily store whatever you need. This allows you to easily change your data storage as you learn more about your project. You can easily add new things as you discover them. Furthermore, if you find you don't need some things anymore, you can just stop storing them, without worrying about losing old data as you would if you delete columns in a relational schema.

As well as handling changes, a schemaless store also makes it easier to deal with **nonuniform data**: data where each record has a different set of fields. A schema puts all rows of a table into a straightjacket, which becomes awkward if you have different kinds of data in different rows. You either end up with lots of columns that are usually null (a sparse table), or you end up with meaningless columns like `custom column 4`. Schemalessness avoids this, allowing each record to contain just what it needs—no more, no less.

Schemalessness is appealing, and it certainly avoids many problems that exist with fixed-schema databases, but it brings some problems of its own. If all you are doing is storing some data and displaying it in a report as a simple list of `fieldName: value` lines then a schema is only going to get in the way. But usually we do with our data more than this, and we do it with programs that need to know that the billing address is called `billingAddress` and not `addressForBilling` and that the `quantify` field is going to be an integer `5` and not `five`.

The vital, if sometimes inconvenient, fact is that whenever we write a program that accesses data, that program almost always relies on some form of implicit schema. Unless it just says something like

Click here to view code image

```
//pseudo code
foreach (Record r in records) {
  foreach (Field f in r.fields) {
    print (f.name, f.value)
  }
}
```

it will assume that certain field names are present and carry data with a certain meaning, and assume something about the type of data stored within that field. Programs are not humans; they cannot read "qty" and infer that that must be the same as "quantity"—at least not unless we specifically program them to do so. So, however schemaless our database is, there is usually an implicit schema present. This **implicit schema** is a set of assumptions about the data's structure in the code that manipulates the data.

Having the implicit schema in the application code results in some problems. It means that in order to understand what data is present you have to dig into the application code. If that code is well structured you should be able to find a clear place

from which to deduce the schema. But there are no guarantees; it all depends on how clear the application code is. Furthermore, the database remains ignorant of the schema—it can't use the schema to help it decide how to store and retrieve data efficiently. It can't apply its own validations upon that data to ensure that different applications don't manipulate data in an inconsistent way.

These are the reasons why relational databases have a fixed schema, and indeed the reasons why most databases have had fixed schemas in the past. Schemas have value, and the rejection of schemas by NoSQL databases is indeed quite startling.

Essentially, a schemaless database shifts the schema into the application code that accesses it. This becomes problematic if multiple applications, developed by different people, access the same database. These problems can be reduced with a couple of approaches. One is to encapsulate all database interaction within a single application and integrate it with other applications using web services. This fits in well with many people's current preference for using web services for integration. Another approach is to clearly delineate different areas of an aggregate for access by different applications. These could be different sections in a document database or different column families an a column-family database.

Although NoSQL fans often criticize relational schemas for having to be defined up front and being inflexible, that's not really true. Relational schemas can be changed at any time with standard SQL commands. If necessary, you can create new columns in an ad-hoc way to store nonuniform data. We have only rarely seen this done, but it worked reasonably well where we have. Most of the time, however, nonuniformity in your data is a good reason to favor a schemaless database.

Schemalessness does have a big impact on changes of a database's structure over time, particularly for more uniform data. Although it's not practiced as widely as it ought to be, changing a relational database's schema can be done in a controlled way. Similarly, you have to exercise control when changing how you store data in a schemaless database so that you can easily access both old and new data. Furthermore, the flexibility that schemalessness gives you only applies within an aggregate—if you need to change your aggregate boundaries, the migration is every bit as complex as it is in the relational case. We'll talk more about database migration later ("Schema Migrations," p. 123).

3.4. Materialized Views

When we talked about aggregate-oriented data models, we stressed their advantages. If you want to access orders, it's useful to have all the data for an order contained in a single aggregate that can be stored and accessed as a unit. But aggregate-orientation has a corresponding disadvantage: What happens if a product manager wants to know how much a particular item has sold over the last couple of weeks? Now the aggregate-orientation works against you, forcing you to potentially read every order in the database to answer the question. You can reduce this burden by building an index on the product, but you're still working against the aggregate structure.

Relational databases have an advantage here because their lack of aggregate structure allows them to support accessing data in different ways. Furthermore, they provide a convenient mechanism that allows you to look at data differently from the way it's stored—views. A view is like a relational table (it is a relation) but it's defined by computation over the base tables. When you access a view, the database computes the data in the view—a handy form of encapsulation.

Views provide a mechanism to hide from the client whether data is derived data or base data—but can't avoid the fact that some views are expensive to compute. To cope with this, **materialized views** were invented, which are views that are computed in advance and cached on disk. Materialized views are effective for data that is read heavily but can stand being somewhat stale.

Although NoSQL databases don't have views, they may have precomputed and cached queries, and they reuse the term "materialized view" to describe them. It's also much more of a central aspect for aggregate-oriented databases than it is for relational systems, since most applications will have to deal with some queries that don't fit well with the aggregate structure. (Often, NoSQL databases create materialized views using a map-reduce computation, which we'll talk about in Chapter 7.)

There are two rough strategies to building a materialized view. The first is the eager approach where you update the materialized view at the same time you update the base data for it. In this case, adding an order would also update the purchase history aggregates for each product. This approach is good when you have more frequent reads of the materialized view than you have writes and you want the materialized views to be as fresh as possible. The application database (p. 7) approach is valuable here as it makes it easier to ensure that any updates to base data also update materialized views.

If you don't want to pay that overhead on each update, you can run batch jobs to update the materialized views at regular intervals. You'll need to understand your business requirements to assess how stale your materialized views can be.

You can build materialized views outside of the database by reading the data, computing the view, and saving it back to the database. More often databases will support building materialized views themselves. In this case, you provide the computation that needs to be done, and the database executes the computation when needed according to some parameters that you configure. This is particularly handy for eager updates of views with incremental map-reduce ("Incremental Map-Reduce," p. 76).

Materialized views can be used within the same aggregate. An order document might include an order summary element that provides summary information about the order so that a query for an order summary does not have to transfer the entire order document. Using different column families for materialized views is a common feature of column-family databases. An advantage of doing this is that it allows you to update the materialized view within the same atomic operation.

3.5. Modeling for Data Access

As mentioned earlier, when modeling data aggregates we need to consider how the data is going to be read as well as what are the side effects on data related to those aggregates.

Let's start with the model where all the data for the customer is embedded using a key-value store (see Figure 3.2).

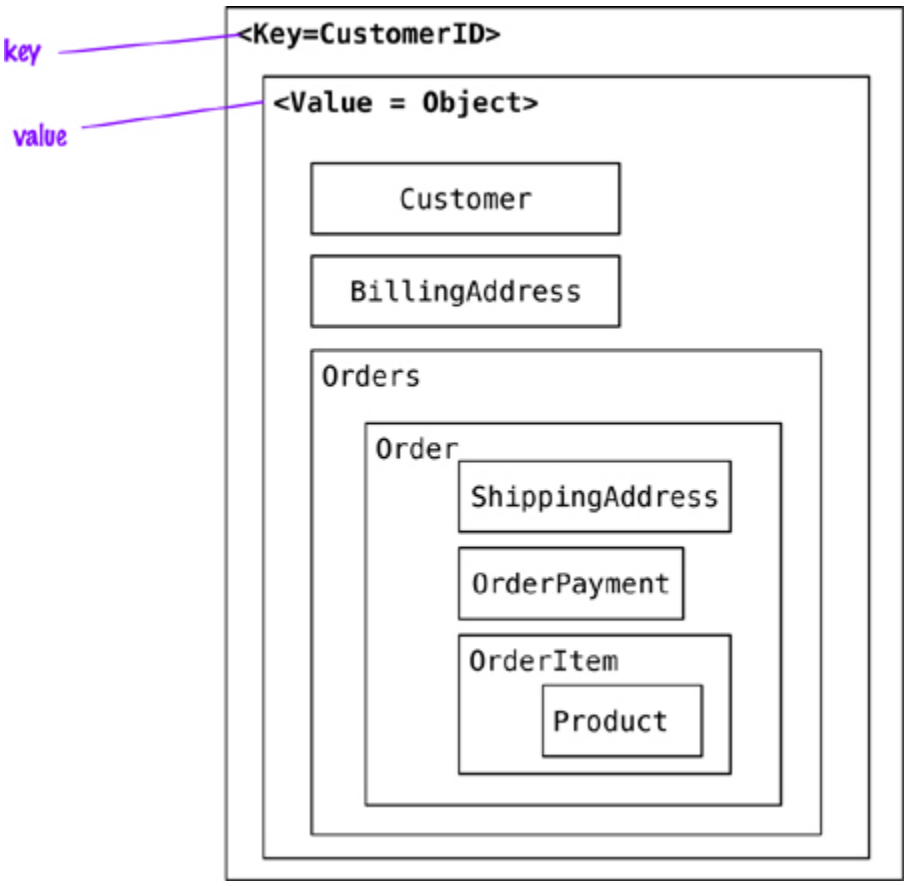

Figure 3.2. Embed all the objects for customer and their orders.

In this scenario, the application can read the customer's information and all the related data by using the key. If the requirements are to read the orders or the products sold in each order, the whole object has to be read and then parsed on the client side to build the results. When references are needed, we could switch to document stores and then query inside the documents, or even change the data for the key-value store to split the value object into `Customer` and `Order` objects and then maintain these objects' references to each other.

With the references (see Figure 3.3), we can now find the orders independently from the `Customer`, and with the `orderId` reference in the `Customer` we can find all `Orders` for the `Customer`. Using aggregates this way allows for read optimization, but we have to push the `orderId` reference into `Customer` every time with a new `Order`.

Click here to view code image

```
# Customer object
{
"customerId": 1,
"customer": {
  "name": "Martin",
  "billingAddress": [{"city": "Chicago"}],
  "payment": [{"type": "debit","ccinfo": "1000-1000-1000-1000"}],
  "orders":[{"orderId":99}]
```

```
 }
}

# Order object
{
"customerId": 1,
"orderId": 99,
"order":{
  "orderDate":"Nov-20-2011",
  "orderItems":[{"productId":27, "price": 32.45}],
   "orderPayment":[{"ccinfo":"1000-1000-1000-1000",
           "txnId":"abelif879rft"}],
   "shippingAddress":{"city":"Chicago"}
   }
}
```

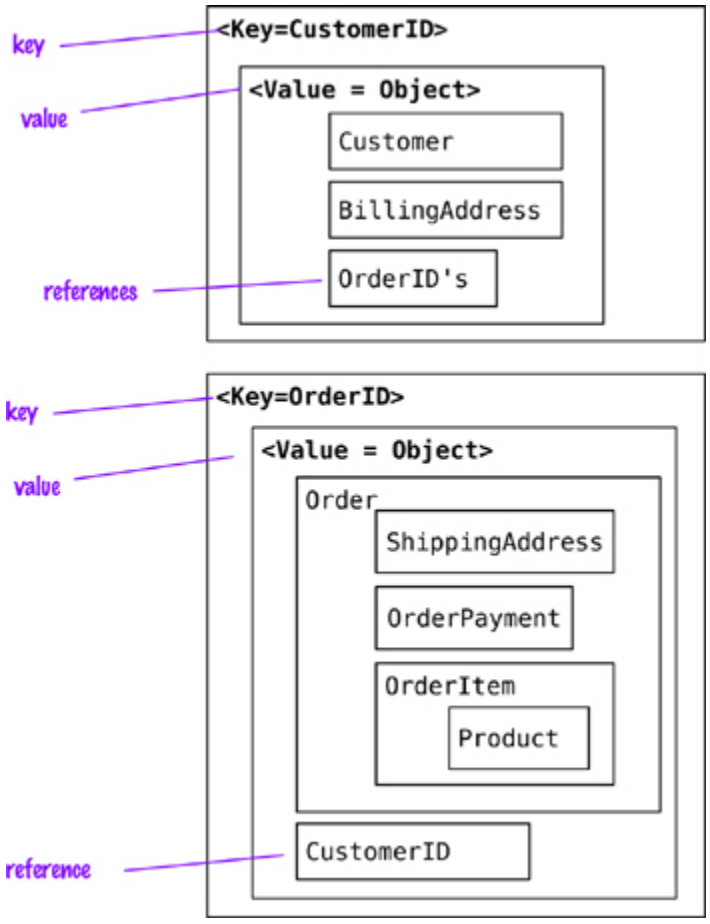

Figure 3.3. `Customer` is stored separately from `Order`.

Aggregates can also be used to obtain analytics; for example, an aggregate update may fill in information on which `Orders` have a given `Product` in them. This denormalization of the data allows for fast access to the data we are interested in and is the basis for **Real Time BI** or **Real Time Analytics** where enterprises don't have to rely on end-of-the-day batch runs to populate data warehouse tables and generate analytics; now they can fill in this type of data, for multiple types of requirements, when the order is placed by the customer.

Click here to view code image

```
{
"itemid":27,
"orders":{99,545,897,678}
}
```

```
{
"itemid":29,
"orders":{199,545,704,819}
}
```

In document stores, since we can query inside documents, removing references to `Orders` from the `Customer` object is possible. This change allows us to not update the `Customer` object when new orders are placed by the `Customer`.

Click here to view code image

```
# Customer object
{
"customerId": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"payment": [
  {"type": "debit",
  "ccinfo": "1000-1000-1000-1000"}
  ]
}
# Order object
{
"orderId": 99,
"customerId": 1,
"orderDate":"Nov-20-2011",
"orderItems":[{"productId":27, "price": 32.45}],
"orderPayment":[{"ccinfo":"1000-1000-1000-1000",
        "txnId":"abelif879rft"}],
"shippingAddress":{"city":"Chicago"}
}
```

Since document data stores allow you to query by attributes inside the document, searches such as “find all orders that include the *Refactoring Databases* product” are possible, but the decision to create an aggregate of items and orders they belong to is not based on the database’s query capability but on the read optimization desired by the application.

When modeling for column-family stores, we have the benefit of the columns being ordered, allowing us to name columns that are frequently used so that they are fetched first. When using the column families to model the data, it is important to remember to do it per your query requirements and not for the purpose of writing; the general rule is to make it easy to query and denormalize the data during write.

As you can imagine, there are multiple ways to model the data; one way is to store the `Customer` and `Order` in different *column-family* families (see Figure 3.4). Here, it is important to note the reference to all the orders placed by the customer are in the `Customer` column family. Similar other denormalizations are generally done so that query (read) performance is improved.

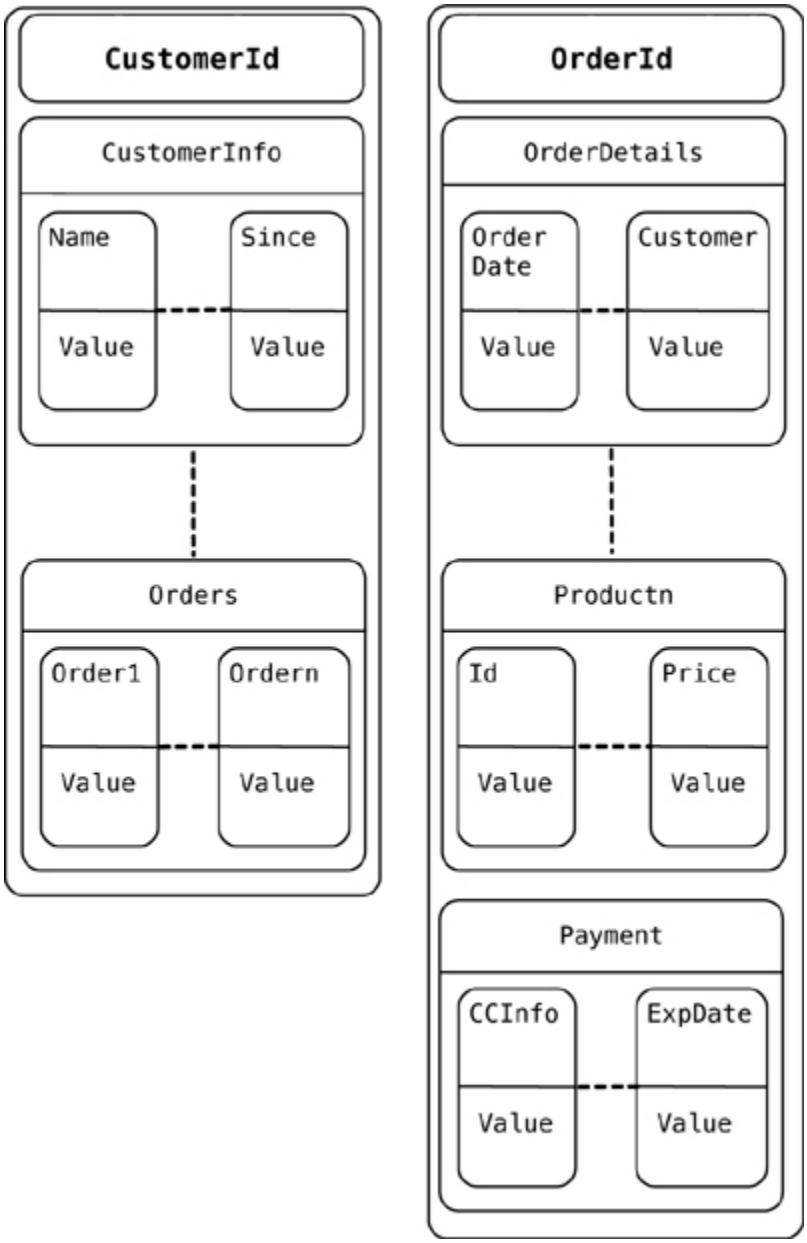

Figure 3.4. Conceptual view into a column data store

When using graph databases to model the same data, we model all objects as nodes and relations within them as relationships; these relationships have types and directional significance.

Each node has independent relationships with other nodes. These relationships have names like *PURCHASED*, *PAID_WITH*, or *BELONGS_TO* (see Figure 3.5); these relationship names let you traverse the graph. Let's say you want to find all the `Customers` who *PURCHASED* a product with the name *Refactoring Database*. All we need to do is query for the product node `Refactoring Databases` and look for all the `Customers` with the incoming *PURCHASED* relationship.

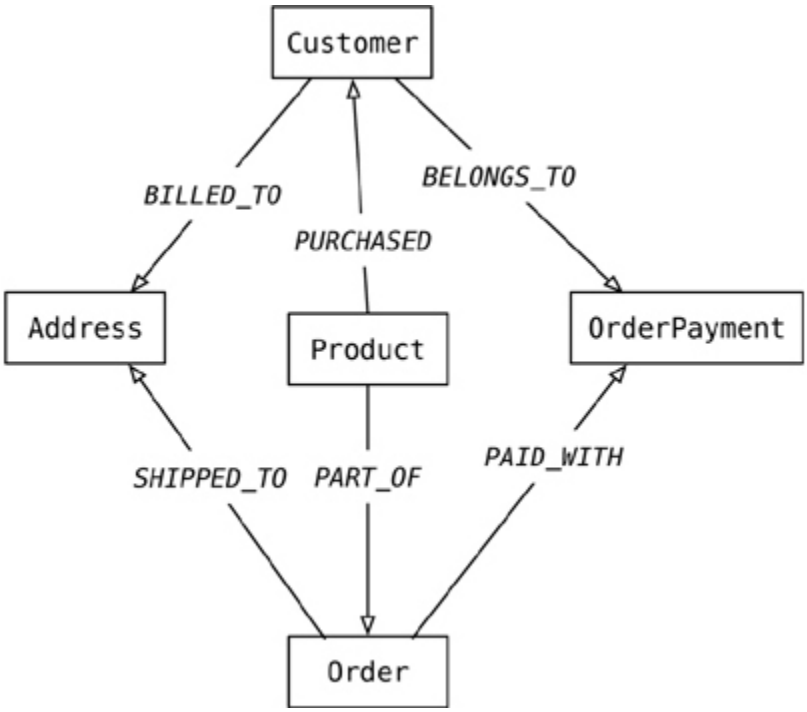

Figure 3.5. Graph model of e-commerce data

This type of relationship traversal is very easy with graph databases. It is especially convenient when you need to use the data to recommend products to users or to find patterns in actions taken by users.

3.6. Key Points

- Aggregate-oriented databases make inter-aggregate relationships more difficult to handle than intra-aggregate relationships.
- Graph databases organize data into node and edge graphs; they work best for data that has complex relationship structures.
- Schemaless databases allow you to freely add fields to records, but there is usually an implicit schema expected by users of the data.
- Aggregate-oriented databases often compute materialized views to provide data organized differently from their primary aggregates. This is often done with map-reduce computations.

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).

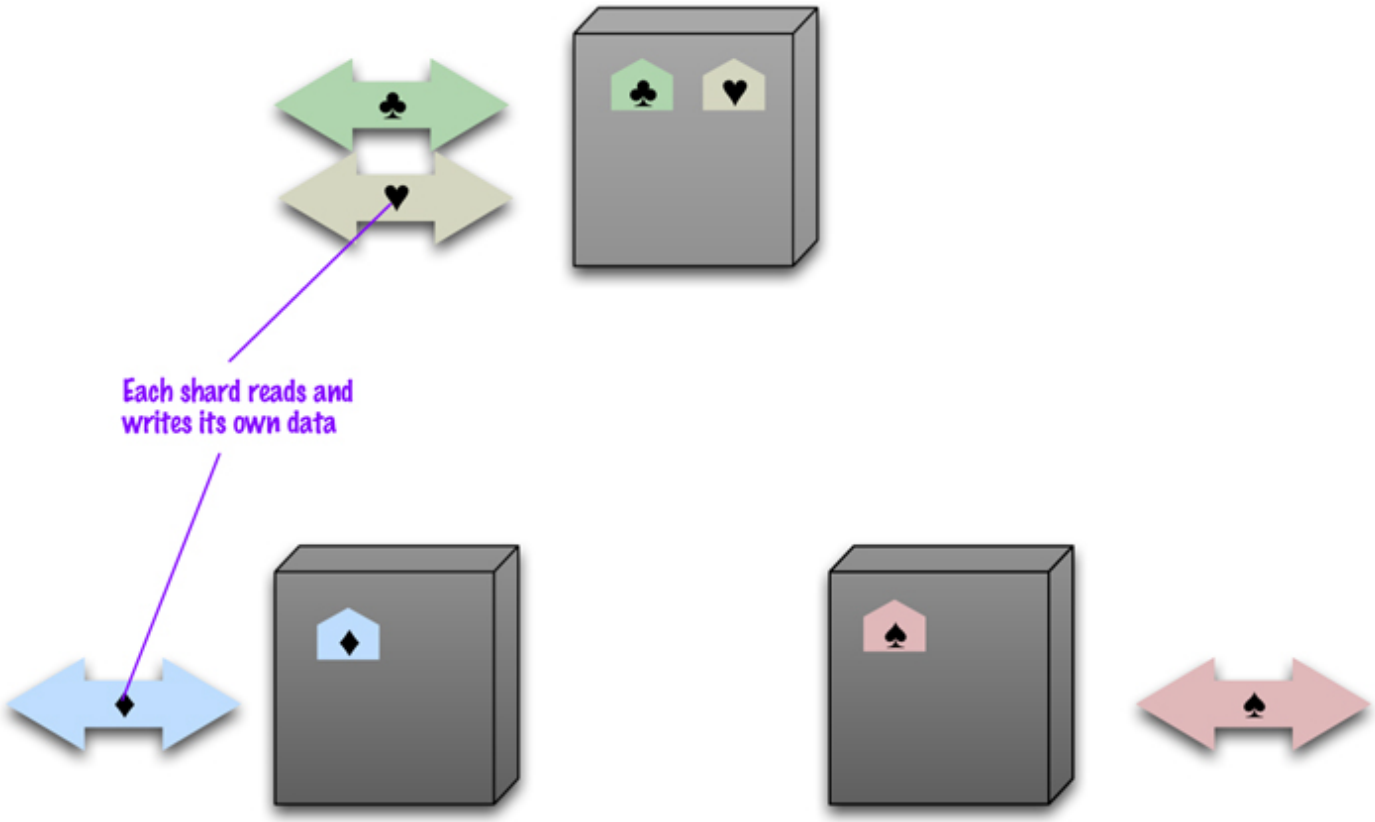

Figure 4.1. Sharding puts different data on separate nodes, each of which does its own reads and writes.

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).

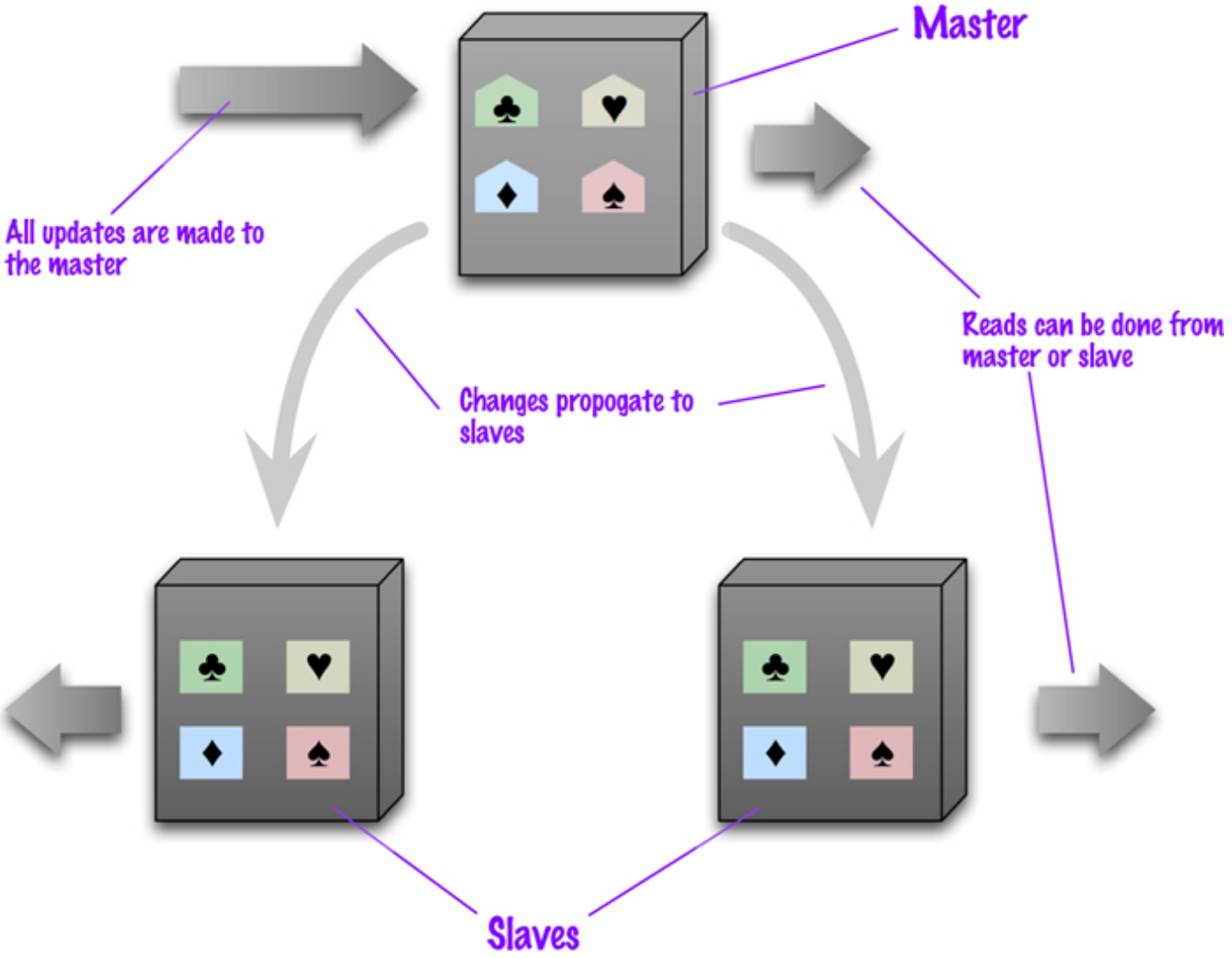

Figure 4.2. Data is replicated from master to slaves. The master services all writes; reads may come from either master or slaves.

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.

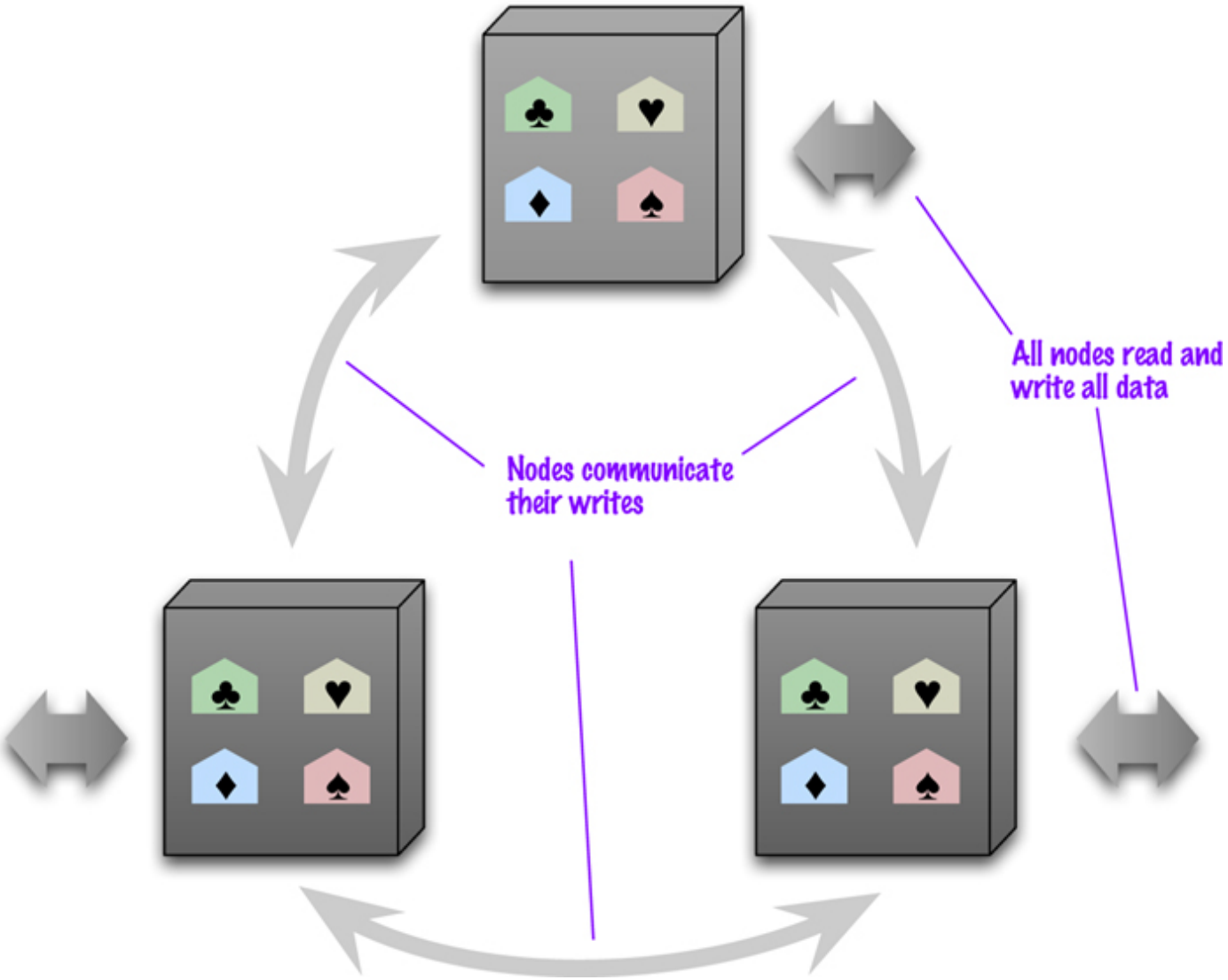

Figure 4.3. Peer-to-peer replication has all nodes applying reads and writes to all the data.

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.

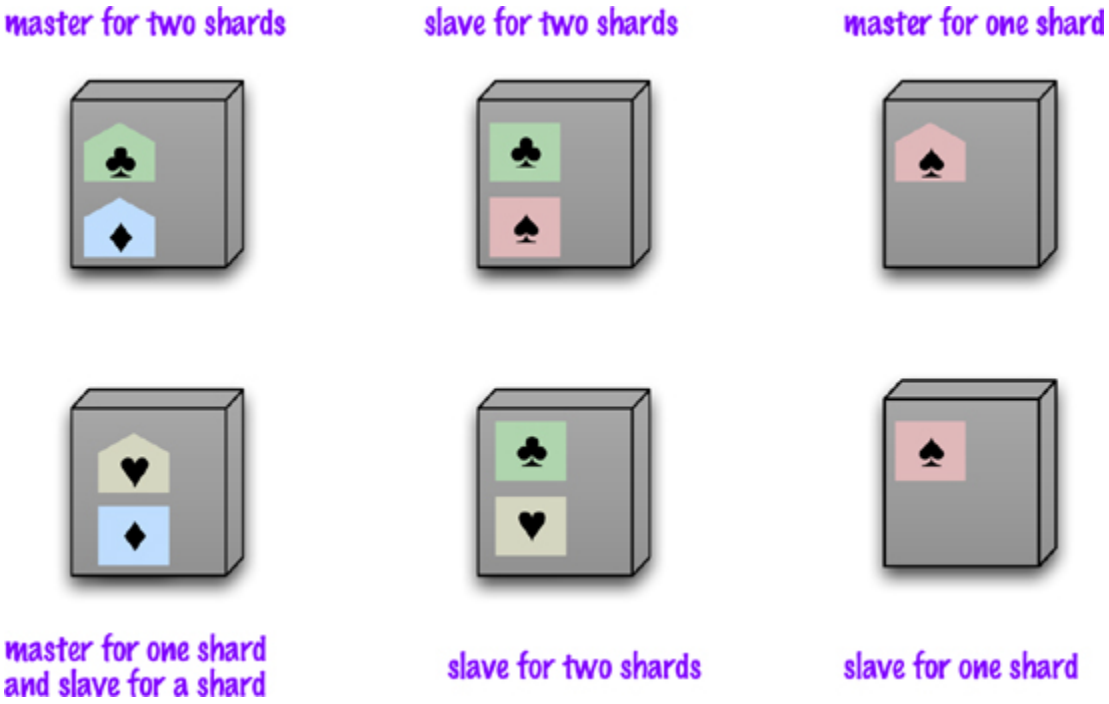

Figure 4.4. Using master-slave replication together with sharding

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).

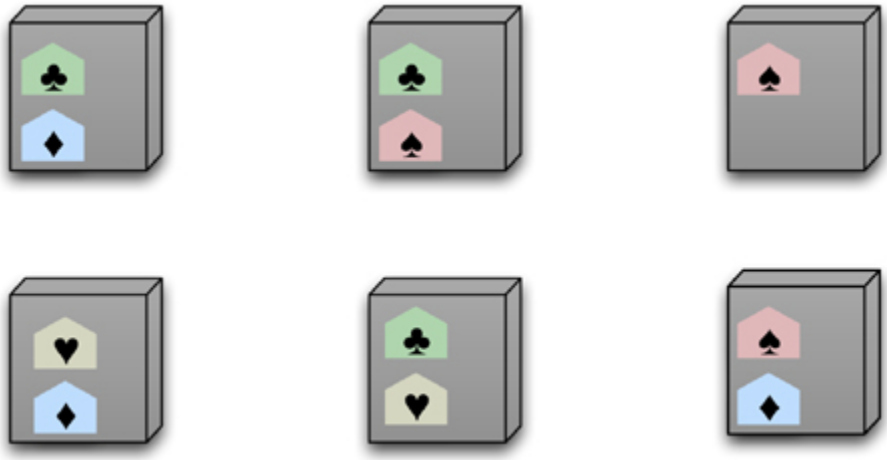

Figure 4.5. Using peer-to-peer replication together with sharding

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.

Chapter 5. Consistency

One of the biggest changes from a centralized relational database to a cluster-oriented NoSQL database is in how you think about consistency. Relational databases try to exhibit **strong consistency** by avoiding all the various inconsistencies that we'll shortly be discussing. Once you start looking at the NoSQL world, phrases such as "CAP theorem" and "eventual consistency" appear, and as soon as you start building something you have to think about what sort of consistency you need for your system.

Consistency comes in various forms, and that one word covers a myriad of ways errors can creep into your life. So we're going to begin by talking about the various shapes consistency can take. After that we'll discuss why you may want to relax consistency (and its big sister, durability).

5.1. Update Consistency

We'll begin by considering updating a telephone number. Coincidentally, Martin and Pramod are looking at the company website and notice that the phone number is out of date. Implausibly, they both have update access, so they both go in at the same time to update the number. To make the example interesting, we'll assume they update it slightly differently, because each uses a slightly different format. This issue is called a **write-write conflict**: two people updating the same data item at the same time.

When the writes reach the server, the server will **serialize** them—decide to apply one, then the other. Let's assume it uses alphabetical order and picks Martin's update first, then Pramod's. Without any concurrency control, Martin's update would be applied and immediately overwritten by Pramod's. In this case Martin's is a **lost update**. Here the lost update is not a big problem, but often it is. We see this as a failure of consistency because Pramod's update was based on the state before Martin's update, yet was applied after it.

Approaches for maintaining consistency in the face of concurrency are often described as pessimistic or optimistic. A **pessimistic** approach works by preventing conflicts from occurring; an **optimistic** approach lets conflicts occur, but detects them and takes action to sort them out. For update conflicts, the most common pessimistic approach is to have write locks, so that in order to change a value you need to acquire a lock, and the system ensures that only one client can get a lock at a time. So Martin and Pramod would both attempt to acquire the write lock, but only Martin (the first one) would succeed. Pramod would then see the result of Martin's write before deciding whether to make his own update.

A common optimistic approach is a **conditional update** where any client that does an update tests the value just before updating it to see if it's changed since his last read. In this case, Martin's update would succeed but Pramod's would fail. The error would let Pramod know that he should look at the value again and decide whether to attempt a further update.

Both the pessimistic and optimistic approaches that we've just described rely on a consistent serialization of the updates. With a single server, this is obvious—it has to choose one, then the other. But if there's more than one server, such as with peer-to-peer replication, then two nodes might apply the updates in a different order, resulting in a different value for the telephone number on each peer. Often, when people talk about concurrency in distributed systems, they talk about sequential consistency—ensuring that all nodes apply operations in the same order.

There is another optimistic way to handle a write-write conflict—save both updates and record that they are in conflict. This approach is familiar to many programmers from version control systems, particularly distributed version control systems that by their nature will often have conflicting commits. The next step again follows from version control: You have to merge the two updates somehow. Maybe you show both values to the user and ask them to sort it out—this is what happens if you update the same contact on your phone and your computer. Alternatively, the computer may be able to perform the merge itself; if it was a phone formatting issue, it may be able to realize that and apply the new number with the standard format. Any automated merge of write-write conflicts is highly domain-specific and needs to be programmed for each particular case.

Often, when people first encounter these issues, their reaction is to prefer pessimistic concurrency because they are determined to avoid conflicts. While in some cases this is the right answer, there is always a tradeoff. Concurrent programming involves a fundamental tradeoff between safety (avoiding errors such as update conflicts) and liveness (responding quickly to clients). Pessimistic approaches often severely degrade the responsiveness of a system to the degree that it becomes unfit for its purpose. This problem is made worse by the danger of errors—pessimistic concurrency often leads to deadlocks, which are hard to prevent and debug.

Replication makes it much more likely to run into write-write conflicts. If different nodes have different copies of some data which can be independently updated, then you'll get conflicts unless you take specific measures to avoid them. Using a single node as the target for all writes for some data makes it much easier to maintain update consistency. Of the distribution models we discussed earlier, all but peer-to-peer replication do this.

5.2. Read Consistency

Having a data store that maintains update consistency is one thing, but it doesn't guarantee that readers of that data store will always get consistent responses to their requests. Let's imagine we have an order with line items and a shipping charge. The shipping charge is calculated based on the line items in the order. If we add a line item, we thus also need to recalculate and update the shipping charge. In a relational database, the shipping charge and line items will be in separate tables. The danger of inconsistency is that Martin adds a line item to his order, Pramod then reads the line items and shipping charge, and then Martin updates the shipping charge. This is an **inconsistent read** or **read-write conflict**: In Figure 5.1 Pramod has done a read in the middle of Martin's write.

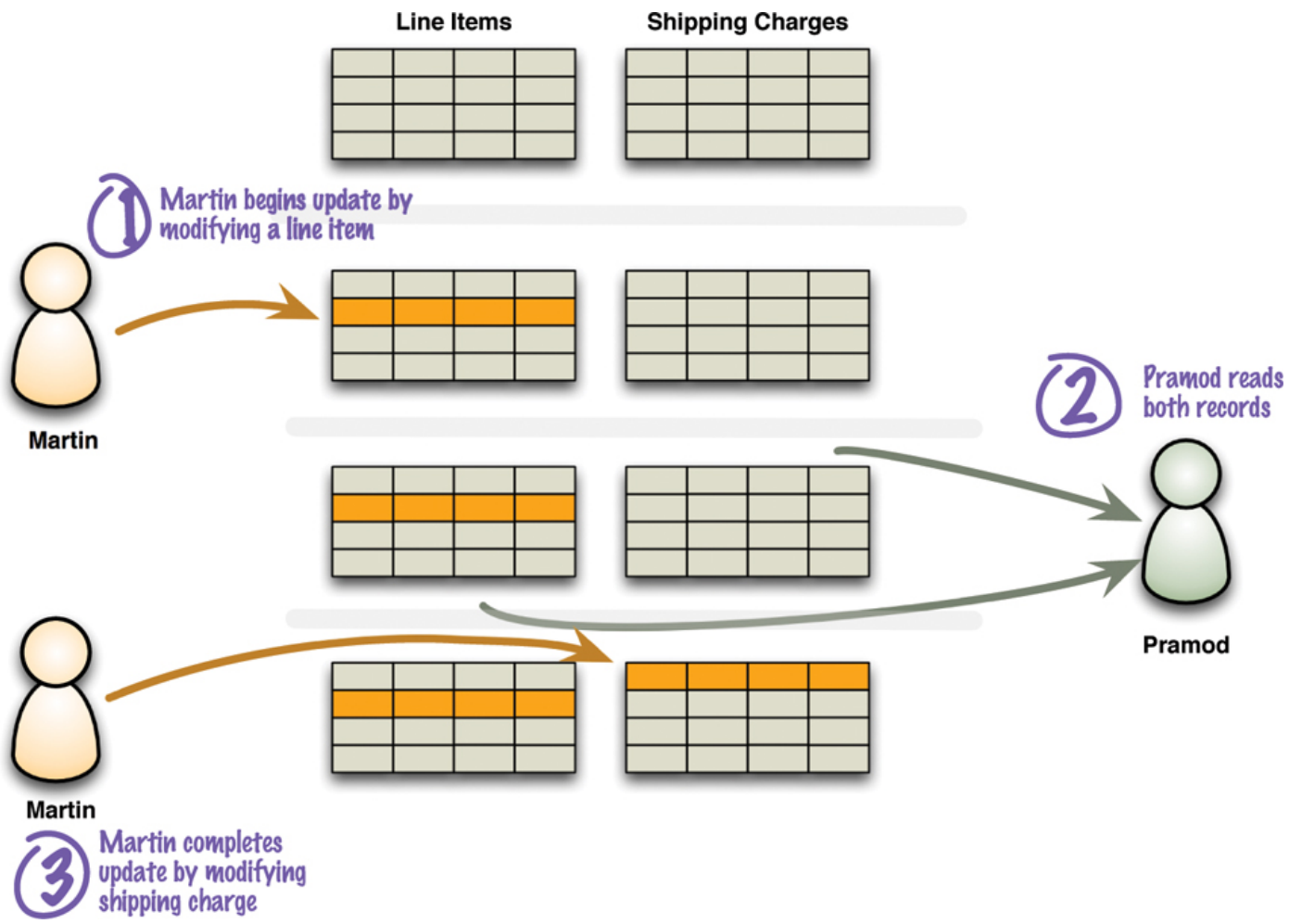

Figure 5.1. A read-write conflict in logical consistency

We refer to this type of consistency as **logical consistency**: ensuring that different data items make sense together. To avoid a logically inconsistent read-write conflict, relational databases support the notion of transactions. Providing Martin wraps his two writes in a transaction, the system guarantees that Pramod will either read both data items before the update or both after the update.

A common claim we hear is that NoSQL databases don't support transactions and thus can't be consistent. Such claim is mostly wrong because it glosses over lots of important details. Our first clarification is that any statement about lack of transactions usually only applies to some NoSQL databases, in particular the aggregate-oriented ones. In contrast, graph databases tend to support ACID transactions just the same as relational databases.

Secondly, aggregate-oriented databases do support atomic updates, but only within a single aggregate. This means that you will have logical consistency within an aggregate but not between aggregates. So in the example, you could avoid running into that inconsistency if the order, the delivery charge, and the line items are all part of a single order aggregate.

Of course not all data can be put in the same aggregate, so any update that affects multiple aggregates leaves open a time when clients could perform an inconsistent read. The length of time an inconsistency is present is called the **inconsistency window**. A NoSQL system may have a quite short inconsistency window: As one data point,

Amazon's documentation says that the inconsistency window for its SimpleDB service is usually less than a second.

This example of a logically inconsistent read is the classic example that you'll see in any book that touches database programming. Once you introduce replication, however, you get a whole new kind of inconsistency. Let's imagine there's one last hotel room for a desirable event. The hotel reservation system runs on many nodes. Martin and Cindy are a couple considering this room, but they are discussing this on the phone because Martin is in London and Cindy is in Boston. Meanwhile Pramod, who is in Mumbai, goes and books that last room. That updates the replicated room availability, but the update gets to Boston quicker than it gets to London. When Martin and Cindy fire up their browsers to see if the room is available, Cindy sees it booked and Martin sees it free. This is another inconsistent read—but it's a breach of a different form of consistency we call **replication consistency**: ensuring that the same data item has the same value when read from different replicas (see Figure 5.2).

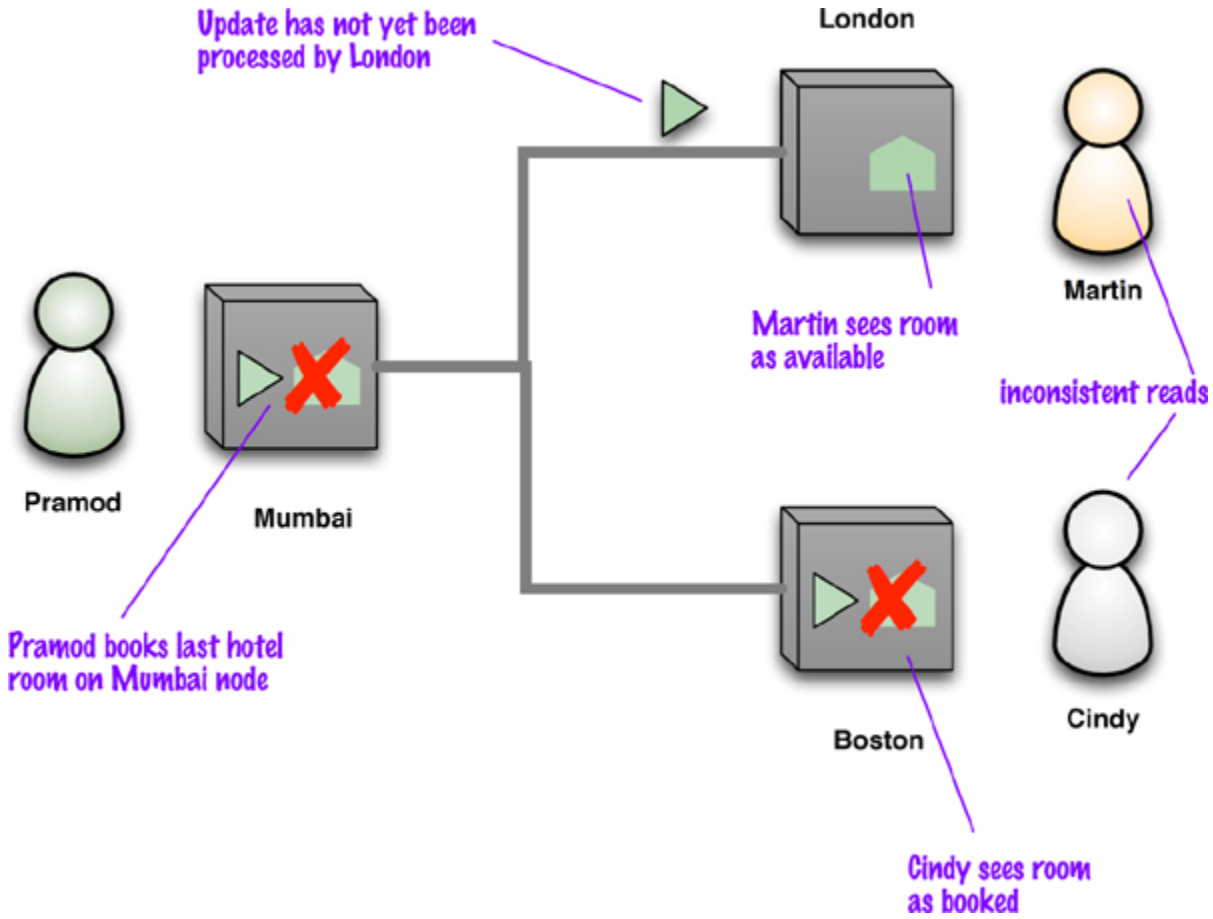

Figure 5.2. An example of replication inconsistency

Eventually, of course, the updates will propagate fully, and Martin will see the room is fully booked. Therefore this situation is generally referred to as **eventually consistent**, meaning that at any time nodes may have replication inconsistencies but, if there are no further updates, eventually all nodes will be updated to the same value. Data that is out of date is generally referred to as **stale**, which reminds us that a cache is another form of replication—essentially following the master-slave distribution model.

Although replication consistency is independent from logical consistency, replication can exacerbate a logical inconsistency by lengthening its inconsistency window. Two different updates on the master may be performed in rapid succession, leaving an

inconsistency window of milliseconds. But delays in networking could mean that the same inconsistency window lasts for much longer on a slave.

Consistency guarantees aren't something that's global to an application. You can usually specify the level of consistency you want with individual requests. This allows you to use weak consistency most of the time when it isn't an issue, but request strong consistency when it is.

The presence of an inconsistency window means that different people will see different things at the same time. If Martin and Cindy are looking at rooms while on a transatlantic call, it can cause confusion. It's more common for users to act independently, and then this is not a problem. But inconsistency windows can be particularly problematic when you get inconsistencies with yourself. Consider the example of posting comments on a blog entry. Few people are going to worry about inconsistency windows of even a few minutes while people are typing in their latest thoughts. Often, systems handle the load of such sites by running on a cluster and load-balancing incoming requests to different nodes. Therein lies a danger: You may post a message using one node, then refresh your browser, but the refresh goes to a different node which hasn't received your post yet—and it looks like your post was lost.

In situations like this, you can tolerate reasonably long inconsistency windows, but you need **read-your-writes consistency** which means that, once you've made an update, you're guaranteed to continue seeing that update. One way to get this in an otherwise eventually consistent system is to provide **session consistency**: Within a user's session there is read-your-writes consistency. This does mean that the user may lose that consistency should their session end for some reason or should the user access the same system simultaneously from different computers, but these cases are relatively rare.

There are a couple of techniques to provide session consistency. A common way, and often the easiest way, is to have a **sticky session**: a session that's tied to one node (this is also called **session affinity**). A sticky session allows you to ensure that as long as you keep read-your-writes consistency on a node, you'll get it for sessions too. The downside is that sticky sessions reduce the ability of the load balancer to do its job.

Another approach for session consistency is to use version stamps ("Version Stamps," p. 61) and ensure every interaction with the data store includes the latest version stamp seen by a session. The server node must then ensure that it has the updates that include that version stamp before responding to a request.

Maintaining session consistency with sticky sessions and master-slave replication can be awkward if you want to read from the slaves to improve read performance but still need to write to the master. One way of handling this is for writes to be sent the slave, who then takes responsibility for forwarding them to the master while maintaining session consistency for its client. Another approach is to switch the session to the master temporarily when doing a write, just long enough that reads are done from the master until the slaves have caught up with the update.

We're talking about replication consistency in the context of a data store, but it's also an important factor in overall application design. Even a simple database system will

have lots of occasions where data is presented to a user, the user cogitates, and then updates that data. It's usually a bad idea to keep a transaction open during user interaction because there's a real danger of conflicts when the user tries to make her update, which leads to such approaches as offline locks [Fowler PoEAA].

5.3. Relaxing Consistency

Consistency is a Good Thing—but, sadly, sometimes we have to sacrifice it. It is always possible to design a system to avoid inconsistencies, but often impossible to do so without making unbearable sacrifices in other characteristics of the system. As a result, we often have to tradeoff consistency for something else. While some architects see this as a disaster, we see it as part of the inevitable tradeoffs involved in system design. Furthermore, different domains have different tolerances for inconsistency, and we need to take this tolerance into account as we make our decisions.

Trading off consistency is a familiar concept even in single-server relational database systems. Here, our principal tool to enforce consistency is the transaction, and transactions can provide strong consistency guarantees. However, transaction systems usually come with the ability to relax isolation levels, allowing queries to read data that hasn't been committed yet, and in practice we see most applications relax consistency down from the highest isolation level (serialized) in order to get effective performance. We most commonly see people using the read-committed transaction level, which eliminates some read-write conflicts but allows others.

Many systems forgo transactions entirely because the performance impact of transactions is too high. We've seen this in a couple different ways. On a small scale, we saw the popularity of MySQL during the days when it didn't support transactions. Many websites liked the high speed of MySQL and were prepared to live without transactions. At the other end of the scale, some very large websites, such as eBay [Pritchett], have had to forgo transactions in order to perform acceptably—this is particularly true when you need to introduce sharding. Even without these constraints, many application builders need to interact with remote systems that can't be properly included within a transaction boundary, so updating outside of transactions is a quite common occurrence for enterprise applications.

5.3.1. The CAP Theorem

In the NoSQL world it's common to refer to the CAP theorem as the reason why you may need to relax consistency. It was originally proposed by Eric Brewer in 2000 [Brewer] and given a formal proof by Seth Gilbert and Nancy Lynch [Lynch and Gilbert] a couple of years later. (You may also hear this referred to as Brewer's Conjecture.)

The basic statement of the CAP theorem is that, given the three properties of Consistency, Availability, and Partition tolerance, you can only get two. Obviously this depends very much on how you define these three properties, and differing opinions have led to several debates on what the real consequences of the CAP theorem are.

Consistency is pretty much as we've defined it so far. **Availability** has a particular meaning in the context of CAP—it means that if you can talk to a node in the cluster, it

can read and write data. That's subtly different from the usual meaning, which we'll explore later. **Partition tolerance** means that the cluster can survive communication breakages in the cluster that separate the cluster into multiple partitions unable to communicate with each other (situation known as a **split brain**, see Figure 5.3).

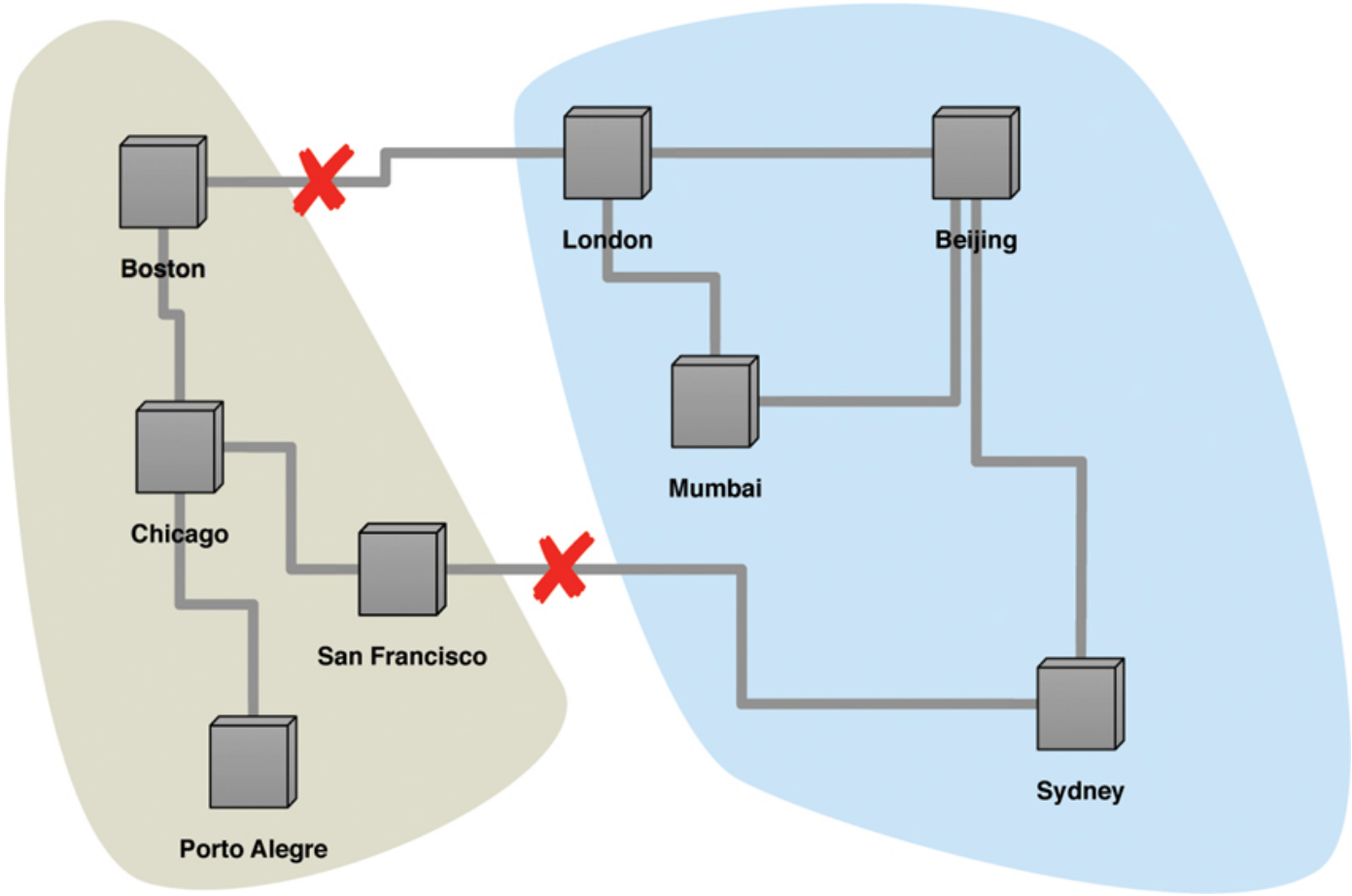

Figure 5.3. With two breaks in the communication lines, the network partitions into two groups.

A single-server system is the obvious example of a CA system—a system that has Consistency and Availability but not Partition tolerance. A single machine can't partition, so it does not have to worry about partition tolerance. There's only one node—so if it's up, it's available. Being up and keeping consistency is reasonable. This is the world that most relational database systems live in.

It is theoretically possible to have a CA cluster. However, this would mean that if a partition ever occurs in the cluster, all the nodes in the cluster would go down so that no client can talk to a node. By the usual definition of "available," this would mean a lack of availability, but this is where CAP's special usage of "availability" gets confusing. CAP defines "availability" to mean "every request received by a nonfailing node in the system must result in a response" [Lynch and Gilbert]. So a failed, unresponsive node doesn't infer a lack of CAP availability.

This does imply that you can build a CA cluster, but you have to ensure it will only partition rarely and completely. This can be done, at least within a data center, but it's usually prohibitively expensive. Remember that in order to bring down all the nodes in a cluster on a partition, you also have to detect the partition in a timely manner—which itself is no small feat.

So clusters have to be tolerant of network partitions. And here is the real point of the CAP theorem. Although the CAP theorem is often stated as “you can only get two out of three,” in practice what it’s saying is that in a system that may suffer partitions, as distributed system do, you have to trade off consistency versus availability. This isn’t a binary decision; often, you can trade off a little consistency to get some availability. The resulting system would be neither perfectly consistent nor perfectly available—but would have a combination that is reasonable for your particular needs.

An example should help illustrate this. Martin and Pramod are both trying to book the last hotel room on a system that uses peer-to-peer distribution with two nodes (London for Martin and Mumbai for Pramod). If we want to ensure consistency, then when Martin tries to book his room on the London node, that node must communicate with the Mumbai node before confirming the booking. Essentially, both nodes must agree on the serialization of their requests. This gives us consistency—but should the network link break, then neither system can book any hotel room, sacrificing availability.

One way to improve availability is to designate one node as the master for a particular hotel and ensure all bookings are processed by that master. Should that master be Mumbai, then Mumbai can still process hotel bookings for that hotel and Pramod will get the last room. If we use master-slave replication, London users can see the inconsistent room information but cannot make a booking and thus cause an update inconsistency. However, users expect that it could happen in this situation—so, again, the compromise works for this particular use case.

This improves the situation, but we still can’t book a room on the London node for the hotel whose master is in Mumbai if the connection goes down. In CAP terminology, this is a failure of availability in that Martin can talk to the London node but the London node cannot update the data. To gain more availability, we might allow both systems to keep accepting hotel reservations even if the network link breaks down. The danger here is that Martin and Pramod book the last hotel room. However, depending on how this hotel operates, that may be fine. Often, travel companies tolerate a certain amount of overbooking in order to cope with no-shows. Conversely, some hotels always keep a few rooms clear even when they are fully booked, in order to be able to swap a guest out of a room with problems or to accommodate a high-status late booking. Some might even cancel the booking with an apology once they detected the conflict—reasoning that the cost of that is less than the cost of losing bookings on network failures.

The classic example of allowing inconsistent writes is the shopping cart, as discussed in Dynamo [Amazon’s Dynamo]. In this case you are always allowed to write to your shopping cart, even if network failures mean you end up with multiple shopping carts. The checkout process can merge the two shopping carts by putting the union of the items from the carts into a single cart and returning that. Almost always that’s the correct answer—but if not, the user gets the opportunity to look at the cart before completing the order.

The lesson here is that although most software developers treat update consistency as The Way Things Must Be, there are cases where you can deal gracefully with inconsistent answers to requests. These situations are closely tied to the domain and

require domain knowledge to know how to resolve. Thus you can't usually look to solve them purely within the development team—you have to talk to domain experts. If you can find a way to handle inconsistent updates, this gives you more options to increase availability and performance. For a shopping cart, it means that shoppers can always shop, and do so quickly. And as Patriotic Americans, we know how vital it is to support Our Retail Destiny.

A similar logic applies to read consistency. If you are trading financial instruments over a computerized exchange, you may not be able to tolerate any data that isn't right up to date. However, if you are posting a news item to a media website, you may be able to tolerate old pages for minutes.

In these cases you need to know how tolerant you are of stale reads, and how long the inconsistency window can be—often in terms of the average length, worst case, and some measure of the distribution for the lengths. Different data items may have different tolerances for staleness, and thus may need different settings in your replication configuration.

Advocates of NoSQL often say that instead of following the ACID properties of relational transactions, NoSQL systems follow the BASE properties (Basically Available, Soft state, Eventual consistency) [Brewer]. Although we feel we ought to mention the BASE acronym here, we don't think it's very useful. The acronym is even more contrived than ACID, and neither "basically available" nor "soft state" have been well defined. We should also stress that when Brewer introduced the notion of BASE, he saw the tradeoff between ACID and BASE as a spectrum, not a binary choice.

We've included this discussion of the CAP theorem because it's often used (and abused) when talking about the tradeoffs involving consistency in distributed databases. However, it's usually better to think not about the tradeoff between consistency and availability but rather between consistency and *latency*. We can summarize much of the discussion about consistency in distribution by saying that we can improve consistency by getting more nodes involved in the interaction, but each node we add increases the response time of that interaction. We can then think of availability as the limit of latency that we're prepared to tolerate; once latency gets too high, we give up and treat the data as unavailable—which neatly fits its definition in the context of CAP.

5.4. Relaxing Durability

So far we've talked about consistency, which is most of what people mean when they talk about the ACID properties of database transactions. The key to Consistency is serializing requests by forming Atomic, Isolated work units. But most people would scoff at relaxing durability—after all, what is the point of a data store if it can lose updates?

As it turns out, there are cases where you may want to trade off some durability for higher performance. If a database can run mostly in memory, apply updates to its in-memory representation, and periodically flush changes to disk, then it may be able to provide substantially higher responsiveness to requests. The cost is that, should the server crash, any updates since the last flush will be lost.

One example of where this tradeoff may be worthwhile is storing user-session state. A big website may have many users and keep temporary information about what each user is doing in some kind of session state. There's a lot of activity on this state, creating lots of demand, which affects the responsiveness of the website. The vital point is that losing the session data isn't too much of a tragedy—it will create some annoyance, but maybe less than a slower website would cause. This makes it a good candidate for nondurable writes. Often, you can specify the durability needs on a call-by-call basis, so that more important updates can force a flush to disk.

Another example of relaxing durability is capturing telemetric data from physical devices. It may be that you'd rather capture data at a faster rate, at the cost of missing the last updates should the server go down.

Another class of durability tradeoffs comes up with replicated data. A failure of **replication durability** occurs when a node processes an update but fails before that update is replicated to the other nodes. A simple case of this may happen if you have a master-slave distribution model where the slaves appoint a new master automatically should the existing master fail. If a master does fail, any writes not passed onto the replicas will effectively become lost. Should the master come back online, those updates will conflict with updates that have happened since. We think of this as a durability problem because you think your update has succeeded since the master acknowledged it, but a master node failure caused it to be lost.

If you're sufficiently confident in bringing the master back online rapidly, this is a reason not to auto-failover to a slave. Otherwise, you can improve replication durability by ensuring that the master waits for some replicas to acknowledge the update before the master acknowledges it to the client. Obviously, however, that will slow down updates and make the cluster unavailable if slaves fail—so, again, we have a tradeoff, depending upon how vital durability is. As with basic durability, it's useful for individual calls to indicate what level of durability they need.

5.5. Quorums

When you're trading off consistency or durability, it's not an all or nothing proposition. The more nodes you involve in a request, the higher is the chance of avoiding an inconsistency. This naturally leads to the question: How many nodes need to be involved to get strong consistency?

Imagine some data replicated over three nodes. You don't need all nodes to acknowledge a write to ensure strong consistency; all you need is two of them—a majority. If you have conflicting writes, only one can get a majority. This is referred to as a **write quorum** and expressed in a slightly pretentious inequality of `W > N/2`, meaning the number of nodes participating in the write (`W`) must be more than the half the number of nodes involved in replication (`N`). The number of replicas is often called the **replication factor**.

Similarly to the write quorum, there is the notion of read quorum: How many nodes you need to contact to be sure you have the most up-to-date change. The read quorum is a bit more complicated because it depends on how many nodes need to confirm a write.

Let's consider a replication factor of 3. If all writes need two nodes to confirm (`W = 2`) then we need to contact at least two nodes to be sure we'll get the latest data. If, however, writes are only confirmed by a single node (`W = 1`) we need to talk to all three nodes to be sure we have the latest updates. In this case, since we don't have a write quorum, we may have an update conflict, but by contacting enough readers we can be sure to detect it. Thus we can get strongly consistent reads even if we don't have strong consistency on our writes.

This relationship between the number of nodes you need to contact for a read (`R`), those confirming a write (`W`), and the replication factor (`N`) can be captured in an inequality: You can have a strongly consistent read if `R + W > N`.

These inequalities are written with a peer-to-peer distribution model in mind. If you have a master-slave distribution, you only have to write to the master to avoid write-write conflicts, and similarly only read from the master to avoid read-write conflicts. With this notation, it is common to confuse the number of nodes in the cluster with the replication factor, but these are often different. I may have 100 nodes in my cluster, but only have a replication factor of 3, with most of the distribution occurring due to sharding.

Indeed most authorities suggest that a replication factor of 3 is enough to have good resilience. This allows a single node to fail while still maintaining quora for reads and writes. If you have automatic rebalancing, it won't take too long for the cluster to create a third replica, so the chances of losing a second replica before a replacement comes up are slight.

The number of nodes participating in an operation can vary with the operation. When writing, we might require quorum for some types of updates but not others, depending on how much we value consistency and availability. Similarly, a read that needs speed but can tolerate staleness should contact less nodes.

Often you may need to take both into account. If you need fast, strongly consistent reads, you could require writes to be acknowledged by all the nodes, thus allowing reads to contact only one (`N = 3`, `W = 3`, `R = 1`). That would mean that your writes are slow, since they have to contact all three nodes, and you would not be able to tolerate losing a node. But in some circumstances that may be the tradeoff to make.

The point to all of this is that you have a range of options to work with and can choose which combination of problems and advantages to prefer. Some writers on NoSQL talk about a simple tradeoff between consistency and availability; we hope you now realize that it's more flexible—and more complicated—than that.

5.6. Further Reading

There are all sorts of interesting blog posts and papers on the Internet about consistency in distributed systems, but the most helpful source for us was [Tanenbaum and Van Steen]. It does an excellent job of organizing much of the fundamentals of distributed systems and is the best place to go if you'd like to delve deeper than we have in this chapter.

As we were finishing this book, *IEEE Computer* had a special issue [IEEE Computer Feb 2012] on the growing influence of the CAP theorem, which is a helpful source of further clarification for this topic.

5.7. Key Points

- Write-write conflicts occur when two clients try to write the same data at the same time. Read-write conflicts occur when one client reads inconsistent data in the middle of another client's write.
- Pessimistic approaches lock data records to prevent conflicts. Optimistic approaches detect conflicts and fix them.
- Distributed systems see read-write conflicts due to some nodes having received updates while other nodes have not. Eventual consistency means that at some point the system will become consistent once all the writes have propagated to all the nodes.
- Clients usually want read-your-writes consistency, which means a client can write and then immediately read the new value. This can be difficult if the read and the write happen on different nodes.
- To get good consistency, you need to involve many nodes in data operations, but this increases latency. So you often have to trade off consistency versus latency.
- The CAP theorem states that if you get a network partition, you have to trade off availability of data versus consistency.
- Durability can also be traded off against latency, particularly if you want to survive failures with replicated data.
- You do not need to contact all replicants to preserve strong consistency with replication; you just need a large enough quorum.

Chapter 6. Version Stamps

Many critics of NoSQL databases focus on the lack of support for transactions. Transactions are a useful tool that helps programmers support consistency. One reason why many NoSQL proponents worry less about a lack of transactions is that aggregate-oriented NoSQL databases do support atomic updates within an aggregate—and aggregates are designed so that their data forms a natural unit of update. That said, it's true that transactional needs are something to take into account when you decide what database to use.

As part of this, it's important to remember that transactions have limitations. Even within a transactional system we still have to deal with updates that require human intervention and usually cannot be run within transactions because they would involve holding a transaction open for too long. We can cope with these using **version stamps**—which turn out to be handy in other situations as well, particularly as we move away from the single-server distribution model.

6.1. Business and System Transactions

The need to support update consistency without transactions is actually a common feature of systems even when they are built on top of transactional databases. When users think about transactions, they usually mean **business transactions**. A business transaction may be something like browsing a product catalog, choosing a bottle of Talisker at a good price, filling in credit card information, and confirming the order. Yet all of this usually won't occur within the **system transaction** provided by the database because this would mean locking the database elements while the user is trying to find their credit card and gets called off to lunch by their colleagues.

Usually applications only begin a system transaction at the end of the interaction with the user, so that the locks are only held for a short period of time. The problem, however, is that calculations and decisions may have been made based on data that's changed. The price list may have updated the price of the Talisker, or someone may have updated the customer's address, changing the shipping charges.

The broad techniques for handling this are offline concurrency [Fowler PoEAA], useful in NoSQL situations too. A particularly useful approach is the Optimistic Offline Lock [Fowler PoEAA], a form of conditional update where a client operation rereads any information that the business transaction relies on and checks that it hasn't changed since it was originally read and displayed to the user. A good way of doing this is to ensure that records in the database contain some form of **version stamp**: a field that changes every time the underlying data in the record changes. When you read the data you keep a note of the version stamp, so that when you write data you can check to see if the version has changed.

You may have come across this technique with updating resources with HTTP [HTTP]. One way of doing this is to use etags. Whenever you get a resource, the server responds with an etag in the header. This etag is an opaque string that indicates the

version of the resource. If you then update that resource, you can use a conditional update by supplying the etag that you got from your last GET. If the resource has changed on the server, the etags won't match and the server will refuse the update, returning a 412 (Precondition Failed) response.

Some databases provide a similar mechanism of conditional update that allows you to ensure updates won't be based on stale data. You can do this check yourself, although you then have to ensure no other thread can run against the resource between your read and your update. (Sometimes this is called a compare-and-set (CAS) operation, whose name comes from the CAS operations done in processors. The difference is that a processor CAS compares a value before setting it, while a database conditional update compares a version stamp of the value.)

There are various ways you can construct your version stamps. You can use a counter, always incrementing it when you update the resource. Counters are useful since they make it easy to tell if one version is more recent than another. On the other hand, they require the server to generate the counter value, and also need a single master to ensure the counters aren't duplicated.

Another approach is to create a GUID, a large random number that's guaranteed to be unique. These use some combination of dates, hardware information, and whatever other sources of randomness they can pick up. The nice thing about GUIDs is that they can be generated by anyone and you'll never get a duplicate; a disadvantage is that they are large and can't be compared directly for recentness.

A third approach is to make a hash of the contents of the resource. With a big enough hash key size, a content hash can be globally unique like a GUID and can also be generated by anyone; the advantage is that they are deterministic—any node will generate the same content hash for same resource data. However, like GUIDs they can't be directly compared for recentness, and they can be lengthy.

A fourth approach is to use the timestamp of the last update. Like counters, they are reasonably short and can be directly compared for recentness, yet have the advantage of not needing a single master. Multiple machines can generate timestamps—but to work properly, their clocks have to be kept in sync. One node with a bad clock can cause all sorts of data corruptions. There's also a danger that if the timestamp is too granular you can get duplicates—it's no good using timestamps of a millisecond precision if you get many updates per millisecond.

You can blend the advantages of these different version stamp schemes by using more than one of them to create a composite stamp. For example, CouchDB uses a combination of counter and content hash. Most of the time this allows version stamps to be compared for recentness, even when you use peer-to-peer replication. Should two peers update at the same time, the combination of the same count and different content hashes makes it easy to spot the conflict.

As well as helping to avoid update conflicts, version stamps are also useful for providing session consistency (p. 52).

6.2. Version Stamps on Multiple Nodes

The basic version stamp works well when you have a single authoritative source for data, such as a single server or master-slave replication. In that case the version stamp is controlled by the master. Any slaves follow the master's stamps. But this system has to be enhanced in a peer-to-peer distribution model because there's no longer a single place to set the version stamps.

If you're asking two nodes for some data, you run into the chance that they may give you different answers. If this happens, your reaction may vary depending on the cause of that difference. It may be that an update has only reached one node but not the other, in which case you can accept the latest (assuming you can tell which one that is). Alternatively, you may have run into an inconsistent update, in which case you need to decide how to deal with that. In this situation, a simple GUID or etag won't suffice, since these don't tell you enough about the relationships.

The simplest form of version stamp is a counter. Each time a node updates the data, it increments the counter and puts the value of the counter into the version stamp. If you have blue and green slave replicas of a single master, and the blue node answers with a version stamp of 4 and the green node with 6, you know that the green's answer is more recent.

In multiple-master cases, we need something fancier. One approach, used by distributed version control systems, is to ensure that all nodes contain a history of version stamps. That way you can see if the blue node's answer is an ancestor of the green's answer. This would either require the clients to hold onto version stamp histories, or the server nodes to keep version stamp histories and include them when asked for data. This also detects an inconsistency, which we would see if we get two version stamps and neither of them has the other in their histories. Although version control systems keep these kinds of histories, they aren't found in NoSQL databases.

A simple but problematic approach is to use timestamps. The main problem here is that it's usually difficult to ensure that all the nodes have a consistent notion of time, particularly if updates can happen rapidly. Should a node's clock get out of sync, it can cause all sorts of trouble. In addition, you can't detect write-write conflicts with timestamps, so it would only work well for the single-master case—and then a counter is usually better.

The most common approach used by peer-to-peer NoSQL systems is a special form of version stamp which we call a vector stamp. In essence, a **vector stamp** is a set of counters, one for each node. A vector stamp for three nodes (blue, green, black) would look something like `[blue: 43, green: 54, black: 12]`. Each time a node has an internal update, it updates its own counter, so an update in the green node would change the vector to `[blue: 43, green: 55, black: 12]`. Whenever two nodes communicate, they synchronize their vector stamps. There are several variations of exactly how this synchronization is done. We're coining the term "vector stamp" as a general term in this book; you'll also come across **vector clocks** and **version vectors**—these are specific forms of vector stamps that differ in how they synchronize.

By using this scheme you can tell if one version stamp is newer than another because the newer stamp will have all its counters greater than or equal to those in the older

stamp. So `[blue: 1, green: 2, black: 5]` is newer than `[blue:1, green: 1, black 5]` since one of its counters is greater. If both stamps have a counter greater than the other, e.g. `[blue: 1, green: 2, black: 5]` and `[blue: 2, green: 1, black: 5]`, then you have a write-write conflict.

There may be missing values in the vector, in which case we use treat the missing value as `0`. So `[blue: 6, black: 2]` would be treated as `[blue: 6, green: 0, black: 2]`. This allows you to easily add new nodes without invalidating the existing vector stamps.

Vector stamps are a valuable tool that spots inconsistencies, but doesn't resolve them. Any conflict resolution will depend on the domain you are working in. This is part of the consistency/latency tradeoff. You either have to live with the fact that network partitions may make your system unavailable, or you have to detect and deal with inconsistencies.

6.3. Key Points

- Version stamps help you detect concurrency conflicts. When you read data, then update it, you can check the version stamp to ensure nobody updated the data between your read and write.
- Version stamps can be implemented using counters, GUIDs, content hashes, timestamps, or a combination of these.
- With distributed systems, a vector of version stamps allows you to detect when different nodes have conflicting updates.

Chapter 7. Map-Reduce

The rise of aggregate-oriented databases is in large part due to the growth of clusters. Running on a cluster means you have to make your tradeoffs in data storage differently than when running on a single machine. Clusters don't just change the rules for data storage—they also change the rules for computation. If you store lots of data on a cluster, processing that data efficiently means you have to think differently about how you organize your processing.

With a centralized database, there are generally two ways you can run the processing logic against it: either on the database server itself or on a client machine. Running it on a client machine gives you more flexibility in choosing a programming environment, which usually makes for programs that are easier to create or extend. This comes at the cost of having to shlep lots of data from the database server. If you need to hit a lot of data, then it makes sense to do the processing on the server, paying the price in programming convenience and increasing the load on the database server.

When you have a cluster, there is good news immediately—you have lots of machines to spread the computation over. However, you also still need to try to reduce the amount of data that needs to be transferred across the network by doing as much processing as you can on the same node as the data it needs.

The map-reduce pattern (a form of *Scatter-Gather* [Hohpe and Woolf]) is a way to organize processing in such a way as to take advantage of multiple machines on a cluster while keeping as much processing and the data it needs together on the same machine. It first gained prominence with Google's MapReduce framework [Dean and Ghemawat]. A widely used open-source implementation is part of the Hadoop project, although several databases include their own implementations. As with most patterns, there are differences in detail between these implementations, so we'll concentrate on the general concept. The name "map-reduce" reveals its inspiration from the map and reduce operations on collections in functional programming languages.

7.1. Basic Map-Reduce

To explain the basic idea, we'll start from an example we've already flogged to death—that of customers and orders. Let's assume we have chosen orders as our aggregate, with each order having line items. Each line item has a product ID, quantity, and the price charged. This aggregate makes a lot of sense as usually people want to see the whole order in one access. We have lots of orders, so we've sharded the dataset over many machines.

However, sales analysis people want to see a product and its total revenue for the last seven days. This report doesn't fit the aggregate structure that we have—which is the downside of using aggregates. In order to get the product revenue report, you'll have to visit every machine in the cluster and examine many records on each machine.

This is exactly the kind of situation that calls for map-reduce. The first stage in a map-reduce job is the map. A map is a function whose input is a single aggregate and

whose output is a bunch of key-value pairs. In this case, the input would be an order. The output would be key-value pairs corresponding to the line items. Each one would have the product ID as the key and an embedded map with the quantity and price as the values (see Figure 7.1).

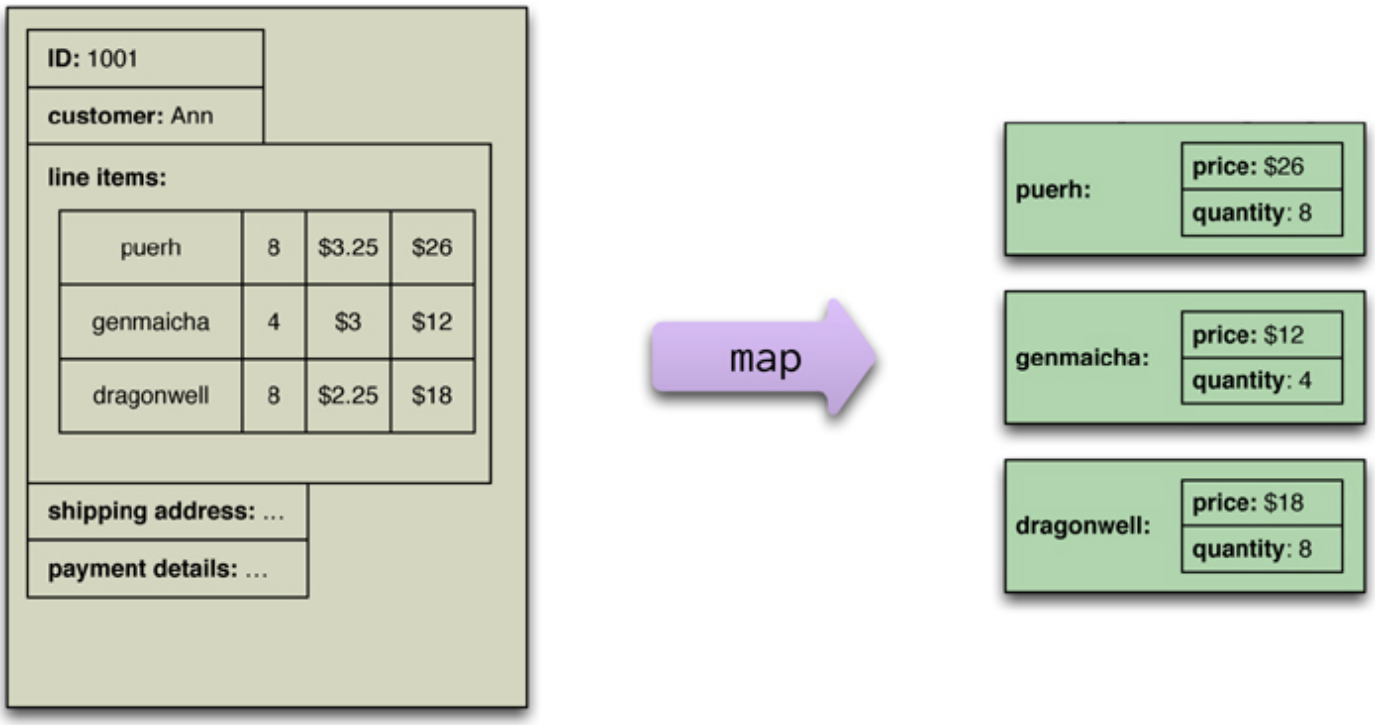

Figure 7.1. A map function reads records from the database and emits key-value pairs.

Each application of the map function is independent of all the others. This allows them to be safely parallelizable, so that a map-reduce framework can create efficient map tasks on each node and freely allocate each order to a map task. This yields a great deal of parallelism and locality of data access. For this example, we are just selecting a value out of the record, but there's no reason why we can't carry out some arbitrarily complex function as part of the map—providing it only depends on one aggregate's worth of data.

A map operation only operates on a single record; the reduce function takes multiple map outputs with the same key and combines their values. So, a map function might yield 1000 line items from orders for "Database Refactoring"; the reduce function would reduce down to one, with the totals for the quantity and revenue. While the map function is limited to working only on data from a single aggregate, the reduce function can use all values emitted for a single key (see Figure 7.2).

Figure 7.2. A reduce function takes several key-value pairs with the same key and aggregates them into one.

The map-reduce framework arranges for map tasks to be run on the correct nodes to process all the documents and for data to be moved to the reduce function. To make it easier to write the reduce function, the framework collects all the values for a single pair and calls the reduce function once with the key and the collection of all the values for that key. So to run a map-reduce job, you just need to write these two functions.

7.2. Partitioning and Combining

In the simplest form, we think of a map-reduce job as having a single reduce function. The outputs from all the map tasks running on the various nodes are concatenated together and sent into the reduce. While this will work, there are things we can do to increase the parallelism and to reduce the data transfer (see Figure 7.3).

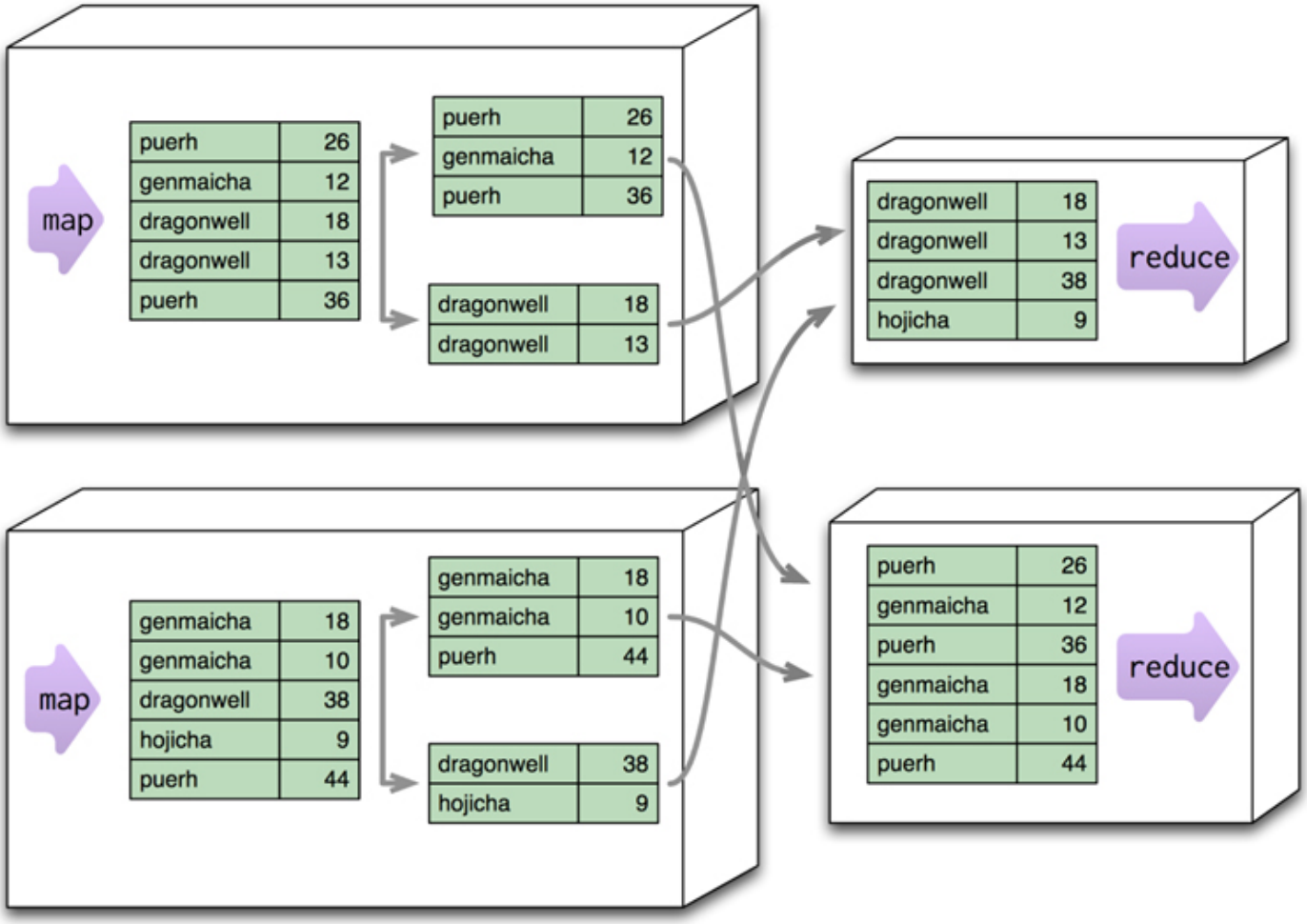

Figure 7.3. Partitioning allows reduce functions to run in parallel on different keys.

The first thing we can do is increase parallelism by partitioning the output of the mappers. Each reduce function operates on the results of a single key. This is a limitation—it means you can't do anything in the reduce that operates across keys—but it's also a benefit in that it allows you to run multiple reducers in parallel. To take advantage of this, the results of the mapper are divided up based the key on each processing node. Typically, multiple keys are grouped together into partitions. The framework then takes the data from all the nodes for one partition, combines it into a single group for that partition, and sends it off to a reducer. Multiple reducers can then operate on the partitions in parallel, with the final results merged together. (This step is also called "shuffling," and the partitions are sometimes referred to as "buckets" or "regions.")

The next problem we can deal with is the amount of data being moved from node to node between the map and reduce stages. Much of this data is repetitive, consisting of multiple key-value pairs for the same key. A combiner function cuts this data down by combining all the data for the same key into a single value (see Figure 7.4). A combiner function is, in essence, a reducer function—indeed, in many cases the same function can be used for combining as the final reduction. The reduce function needs a special shape for this to work: Its output must match its input. We call such a function a **combinable reducer**.

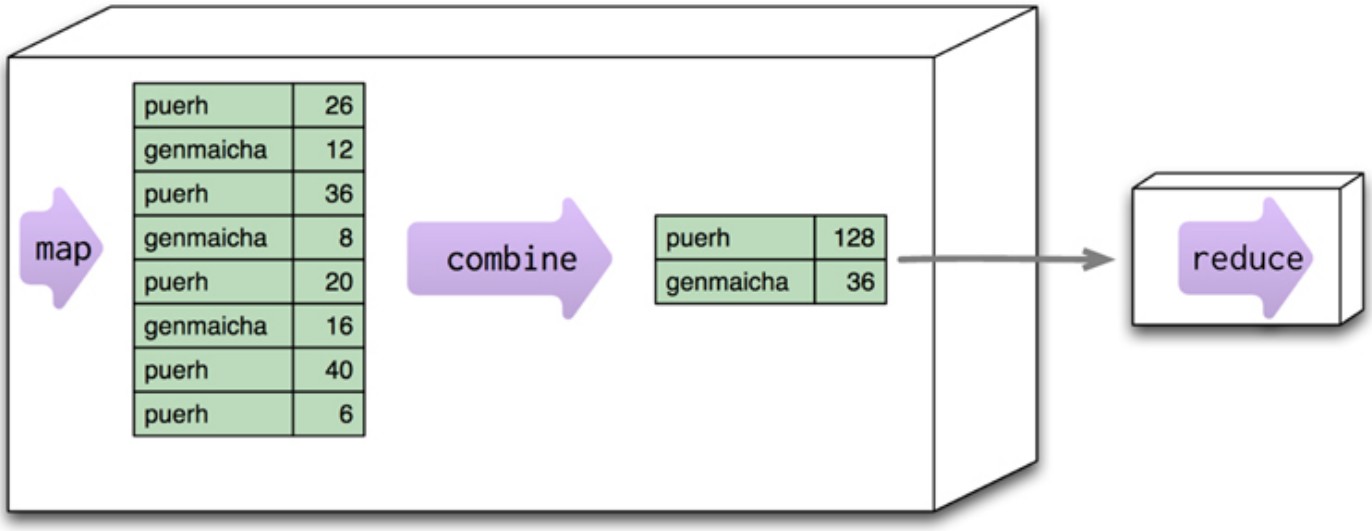

Figure 7.4. Combining reduces data before sending it across the network.

Not all reduce functions are combinable. Consider a function that counts the number of unique customers for a particular product. The map function for such an operation would need to emit the product and the customer. The reducer can then combine them and count how many times each customer appears for a particular product, emitting the product and the count (see Figure 7.5). But this reducer's output is different from its input, so it can't be used as a combiner. You can still run a combining function here: one that just eliminates duplicate product-customer pairs, but it will be different from the final reducer.

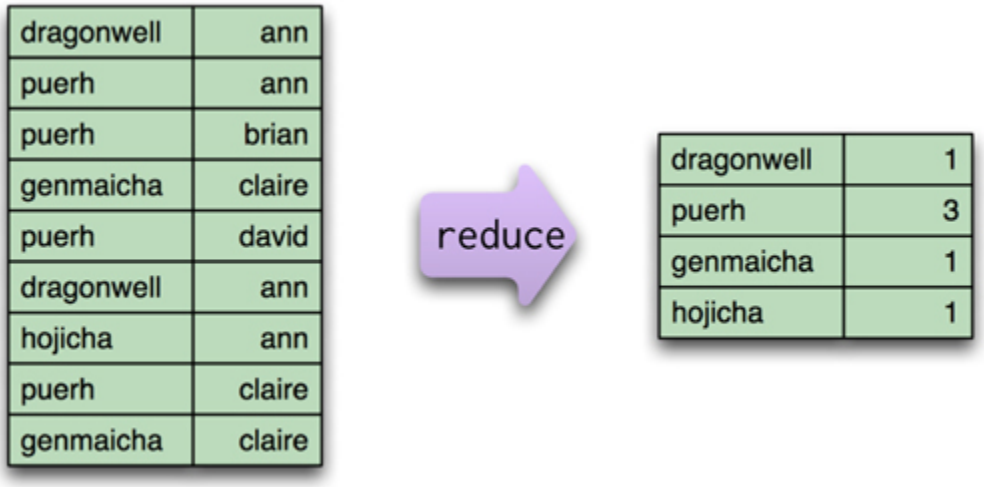

Figure 7.5. This reduce function, which counts how many unique customers order a particular tea, is not combinable.

When you have combining reducers, the map-reduce framework can safely run not only in parallel (to reduce different partitions), but also in series to reduce the same partition at different times and places. In addition to allowing combining to occur on a node before data transmission, you can also start combining before mappers have finished. This provides a good bit of extra flexibility to the map-reduce processing. Some map-reduce frameworks require all reducers to be combining reducers, which maximizes this flexibility. If you need to do a noncombining reducer with one of these frameworks, you'll need to separate the processing into pipelined map-reduce steps.

7.3. Composing Map-Reduce Calculations

The map-reduce approach is a way of thinking about concurrent processing that trades off flexibility in how you structure your computation for a relatively straightforward model for parallelizing the computation over a cluster. Since it's a tradeoff, there are constraints on what you can do in your calculations. Within a map task, you can only operate on a single aggregate. Within a reduce task, you can only operate on a single key. This means you have to think differently about structuring your programs so they work well within these constraints.

One simple limitation is that you have to structure your calculations around operations that fit in well with the notion of a reduce operation. A good example of this is calculating averages. Let's consider the kind of orders we've been looking at so far; suppose we want to know the average ordered quantity of each product. An important property of averages is that they are not composable—that is, if I take two groups of orders, I can't combine their averages alone. Instead, I need to take total amount and the count of orders from each group, combine those, and then calculate the average from the combined sum and count (see Figure 7.6).

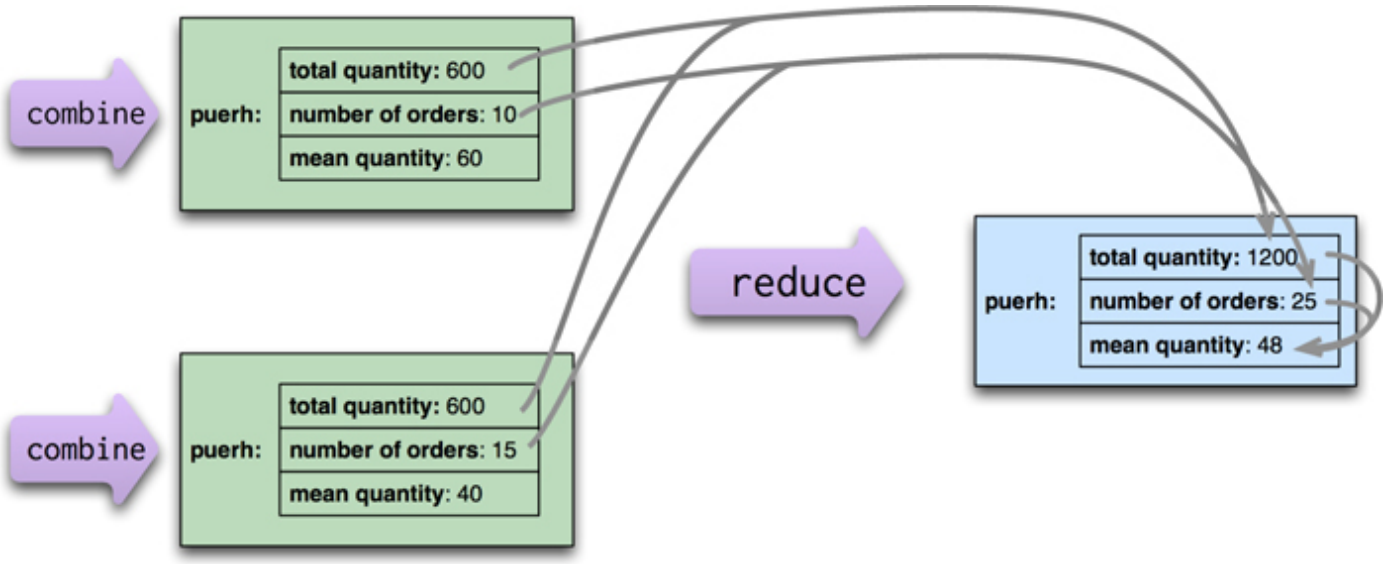

Figure 7.6. When calculating averages, the sum and count can be combined in the reduce calculation, but the average must be calculated from the combined sum and count.

This notion of looking for calculations that reduce neatly also affects how we do counts. To make a count, the mapping function will emit count fields with a value of `1`, which can be summed to get a total count (see Figure 7.7).

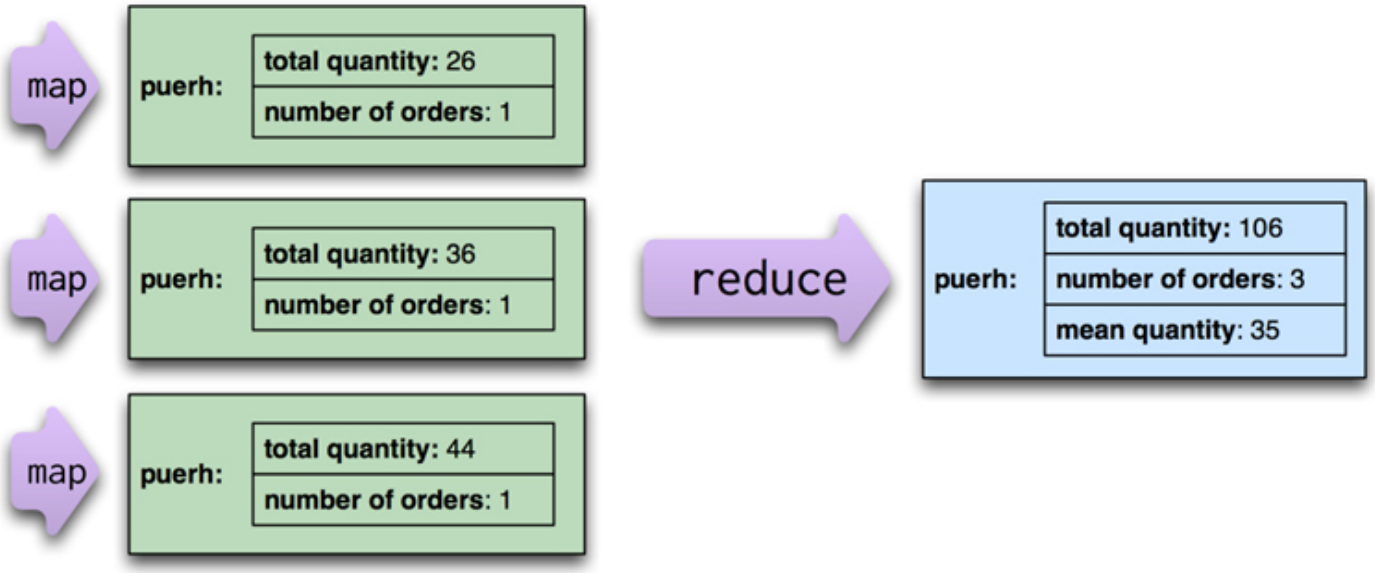

Figure 7.7. When making a count, each map emits 1, which can be summed to get a total.

7.3.1. A Two Stage Map-Reduce Example

As map-reduce calculations get more complex, it's useful to break them down into stages using a pipes-and-filters approach, with the output of one stage serving as input to the next, rather like the pipelines in UNIX.

Consider an example where we want to compare the sales of products for each month in 2011 to the prior year. To do this, we'll break the calculations down into two stages. The first stage will produce records showing the aggregate figures for a single product in a single month of the year. The second stage then uses these as inputs and produces the result for a single product by comparing one month's results with the same month in the prior year (see Figure 7.8).

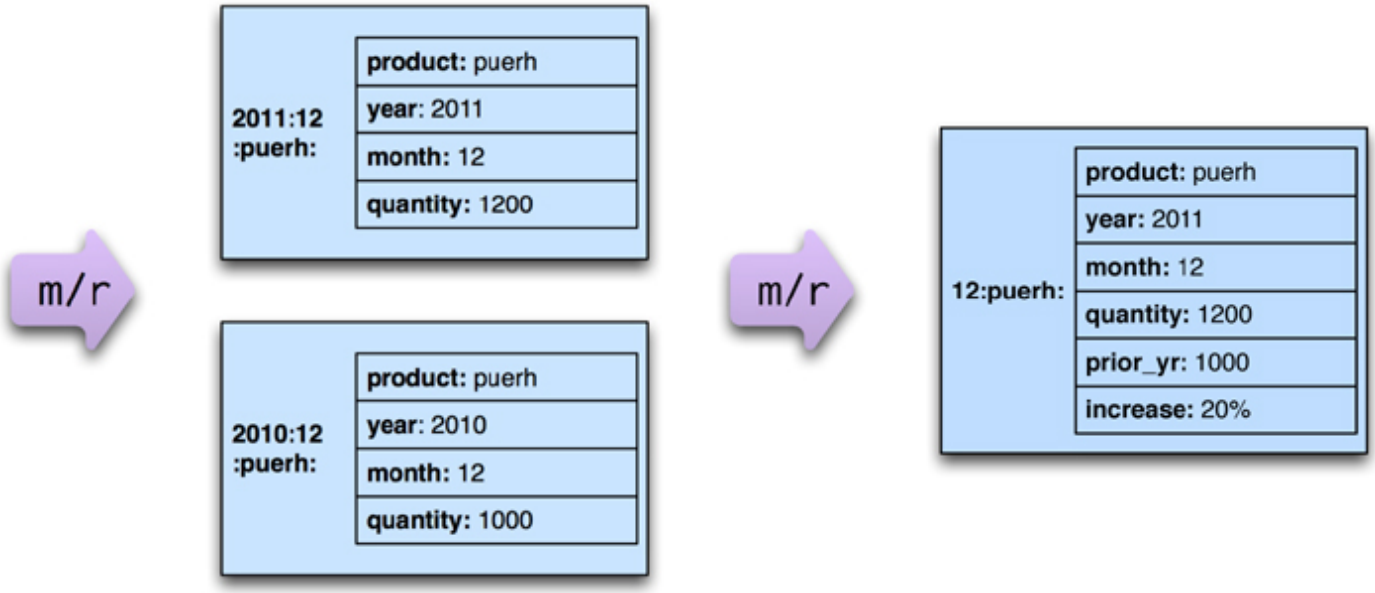

Figure 7.8. A calculation broken down into two map-reduce steps, which will be expanded in the next three figures

A first stage (Figure 7.9) would read the original order records and output a series of key-value pairs for the sales of each product per month.

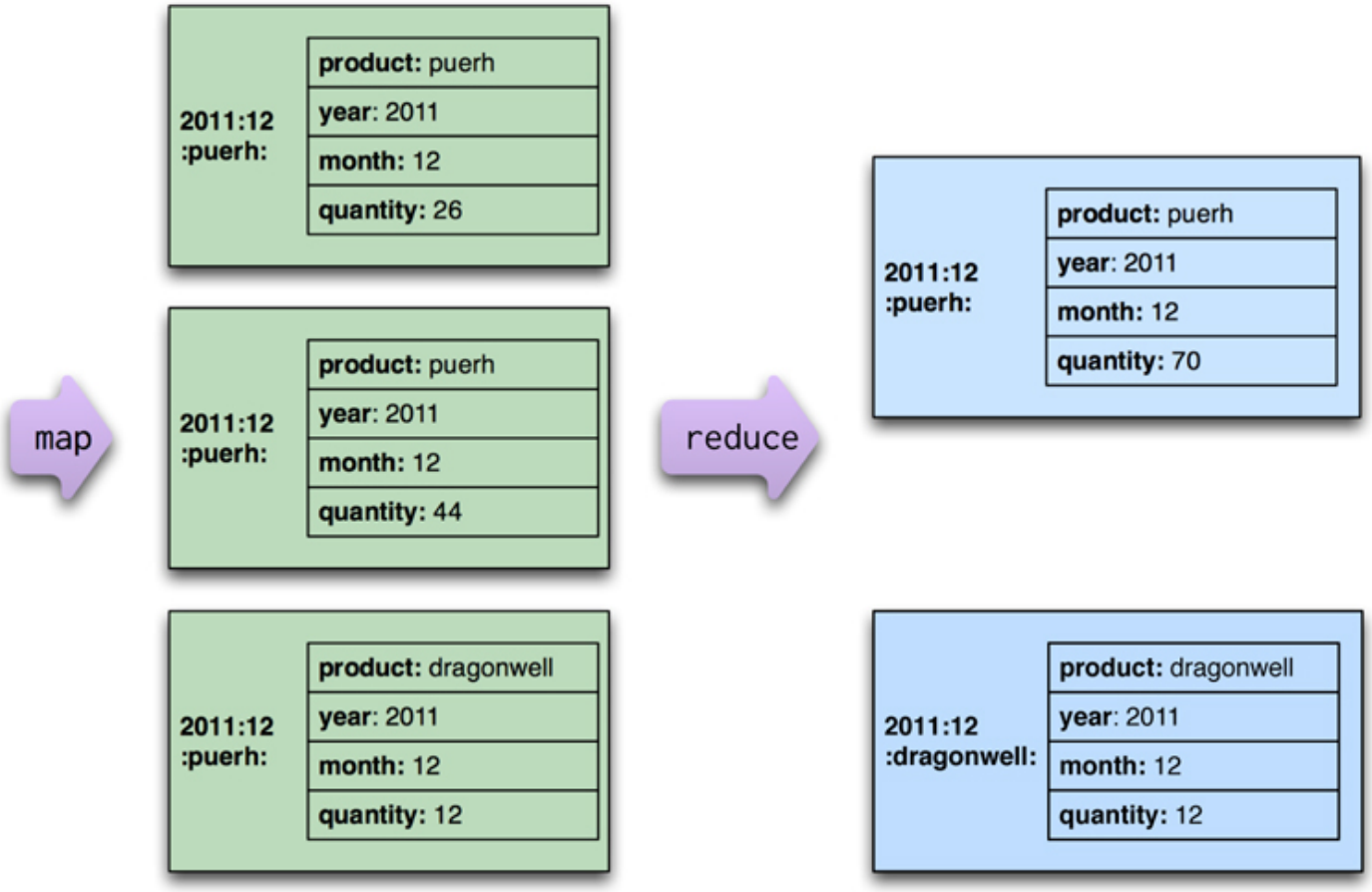

Figure 7.9. Creating records for monthly sales of a product

This stage is similar to the map-reduce examples we've seen so far. The only new feature is using a composite key so that we can reduce records based on the values of multiple fields.

The second-stage mappers (Figure 7.10) process this output depending on the year. A 2011 record populates the current year quantity while a 2010 record populates a prior year quantity. Records for earlier years (such as 2009) don't result in any mapping output being emitted.

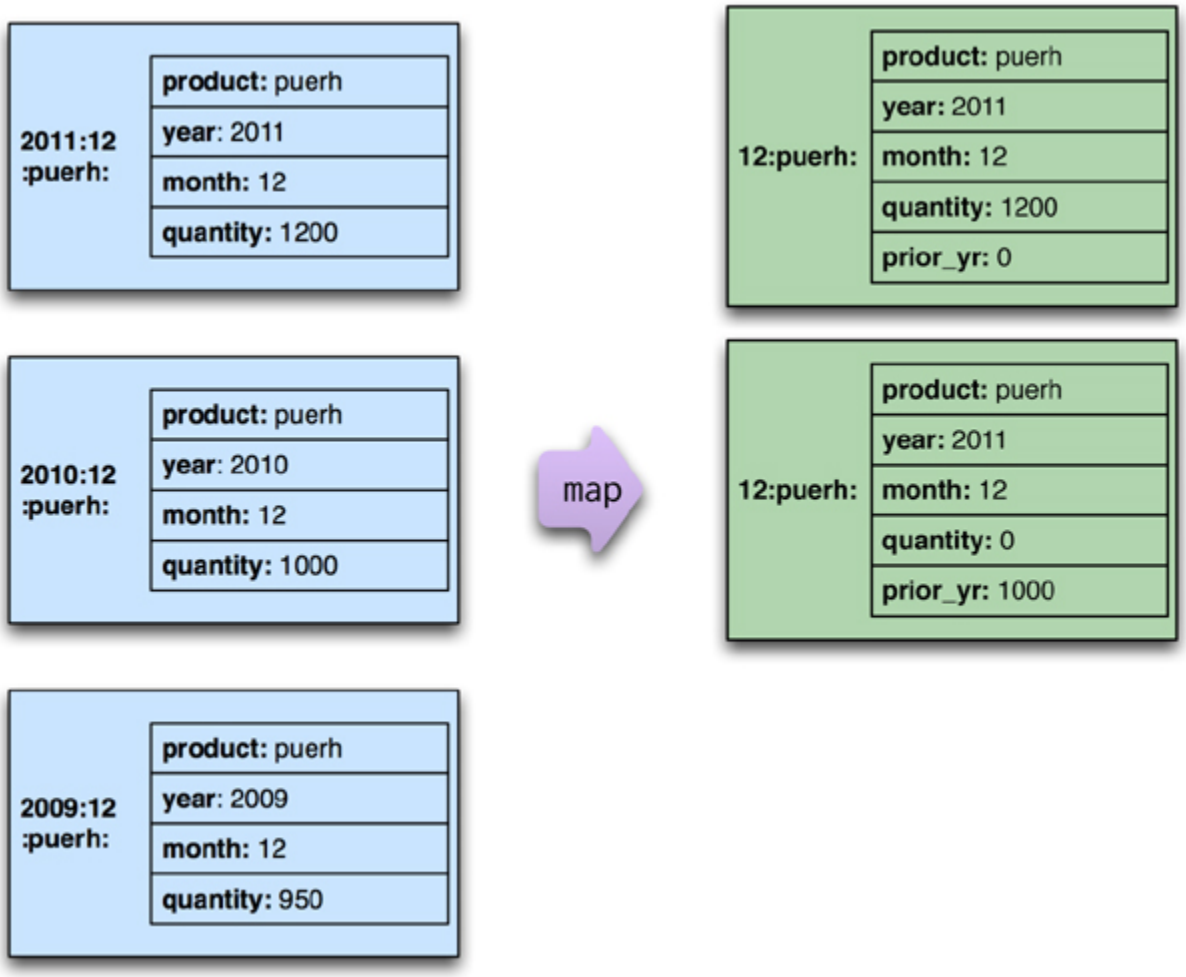

Figure 7.10. The second stage mapper creates base records for year-on-year comparisons.

The reduce in this case (Figure 7.11) is a merge of records, where combining the values by summing allows two different year outputs to be reduced to a single value (with a calculation based on the reduced values thrown in for good measure).

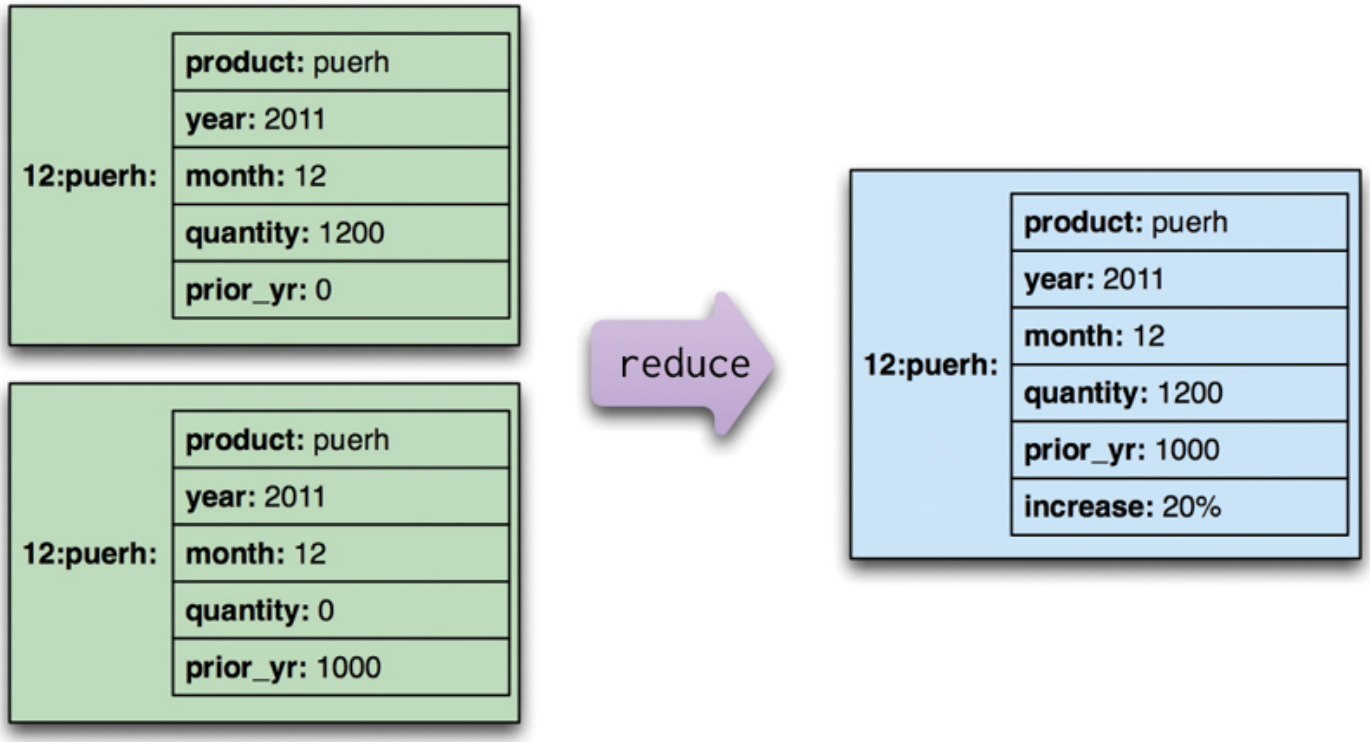

Figure 7.11. The reduction step is a merge of incomplete records.

Decomposing this report into multiple map-reduce steps makes it easier to write. Like many transformation examples, once you've found a transformation framework that makes it easy to compose steps, it's usually easier to compose many small steps together than try to cram heaps of logic into a single step.

Another advantage is that the intermediate output may be useful for different outputs too, so you can get some reuse. This reuse is important as it saves time both in programming and in execution. The intermediate records can be saved in the data store, forming a materialized view ("Materialized Views," p. 30). Early stages of map-reduce operations are particularly valuable to save since they often represent the heaviest amount of data access, so building them once as a basis for many downstream uses saves a lot of work. As with any reuse activity, however, it's important to build them out of experience with real queries, as speculative reuse rarely fulfills its promise. So it's important to look at the forms of various queries as they are built and factor out the common parts of the calculations into materialized views.

Map-reduce is a pattern that can be implemented in any programming language. However, the constraints of the style make it a good fit for languages specifically designed for map-reduce computations. Apache Pig [Pig], an offshoot of the Hadoop [Hadoop] project, is a language specifically built to make it easy to write map-reduce programs. It certainly makes it much easier to work with Hadoop than the underlying Java libraries. In a similar vein, if you want to specify map-reduce programs using an SQL-like syntax, there is hive [Hive], another Hadoop offshoot.

The map-reduce pattern is important to know about even outside of the context of NoSQL databases. Google's original map-reduce system operated on files stored on a distributed file system—an approach that's used by the open-source Hadoop project. While it takes some thought to get used to the constraints of structuring computations in map-reduce steps, the result is a calculation that is inherently well-suited to running on a cluster. When dealing with high volumes of data, you need to take a cluster-oriented approach. Aggregate-oriented databases fit well with this style of calculation. We think that in the next few years many more organizations will be processing the volumes of data that demand a cluster-oriented solution—and the map-reduce pattern will see more and more use.

7.3.2. Incremental Map-Reduce

The examples we've discussed so far are complete map-reduce computations, where we start with raw inputs and create a final output. Many map-reduce computations take a while to perform, even with clustered hardware, and new data keeps coming in which means we need to rerun the computation to keep the output up to date. Starting from scratch each time can take too long, so often it's useful to structure a map-reduce computation to allow incremental updates, so that only the minimum computation needs to be done.

The map stages of a map-reduce are easy to handle incrementally—only if the input data changes does the mapper need to be rerun. Since maps are isolated from each other, incremental updates are straightforward.

The more complex case is the reduce step, since it pulls together the outputs from many maps and any change in the map outputs could trigger a new reduction. This recomputation can be lessened depending on how parallel the reduce step is. If we are partitioning the data for reduction, then any partition that's unchanged does not need to be re-reduced. Similarly, if there's a combiner step, it doesn't need to be rerun if its source data hasn't changed.

If our reducer is combinable, there's some more opportunities for computation avoidance. If the changes are additive—that is, if we are only adding new records but are not changing or deleting any old records—then we can just run the reduce with the existing result and the new additions. If there are destructive changes, that is updates and deletes, then we can avoid some recomputation by breaking up the reduce operation into steps and only recalculating those steps whose inputs have changed—essentially, using a *Dependency Network* [Fowler DSL] to organize the computation.

The map-reduce framework controls much of this, so you have to understand how a specific framework supports incremental operation.

7.4. Further Reading

If you're going to use map-reduce calculations, your first port of call will be the documentation for the particular database you are using. Each database has its own approach, vocabulary, and quirks, and that's what you'll need to be familiar with. Beyond that, there is a need to capture more general information on how to structure map-reduce jobs to maximize maintainability and performance. We don't have any specific books to point to yet, but we suspect that a good though easily overlooked source are books on Hadoop. Although Hadoop is not a database, it's a tool that uses map-reduce heavily, so writing an effective map-reduce task with Hadoop is likely to be useful in other contexts (subject to the changes in detail between Hadoop and whatever systems you're using).

7.5. Key Points

- Map-reduce is a pattern to allow computations to be parallelized over a cluster.
- The map task reads data from an aggregate and boils it down to relevant key-value pairs. Maps only read a single record at a time and can thus be parallelized and run on the node that stores the record.
- Reduce tasks take many values for a single key output from map tasks and summarize them into a single output. Each reducer operates on the result of a single key, so it can be parallelized by key.
- Reducers that have the same form for input and output can be combined into pipelines. This improves parallelism and reduces the amount of data to be transferred.
- Map-reduce operations can be composed into pipelines where the output of one reduce is the input to another operation's map.
- If the result of a map-reduce computation is widely used, it can be stored as a materialized view.

- Materialized views can be updated through incremental map-reduce operations that only compute changes to the view instead of recomputing everything from scratch.