MODULE 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).

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

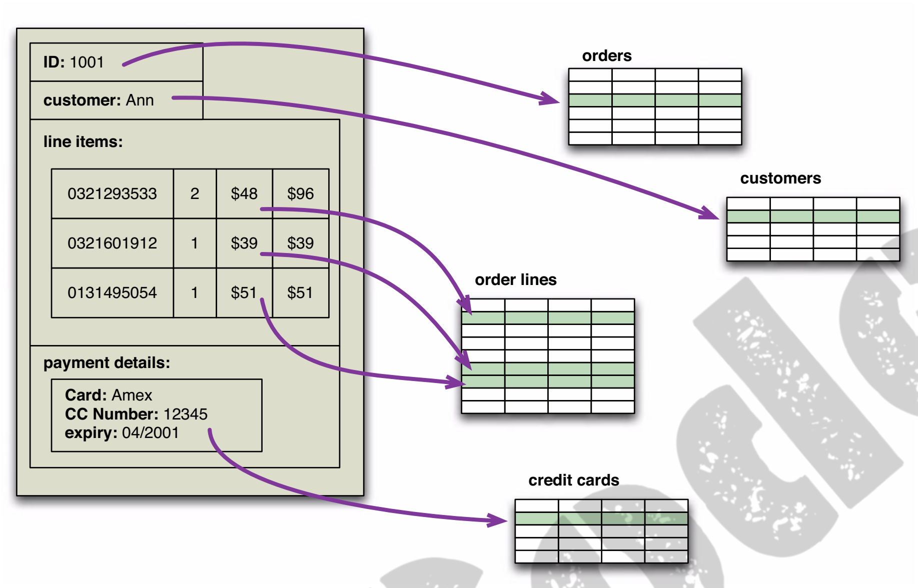


Figure 1.1 *An order, which looks like a single aggregate structure in the UI, is split into many rows from many tables in a relational database*

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 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.

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 2.2 presents some sample data for this 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).

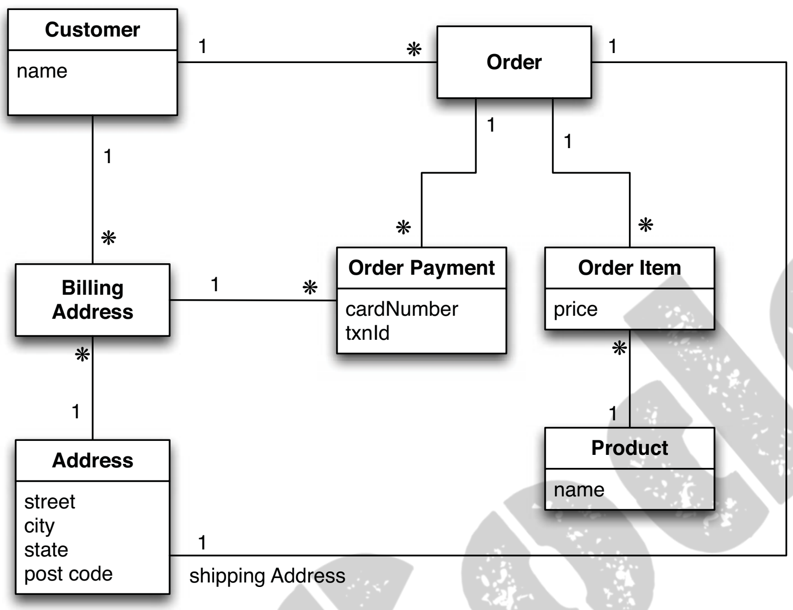
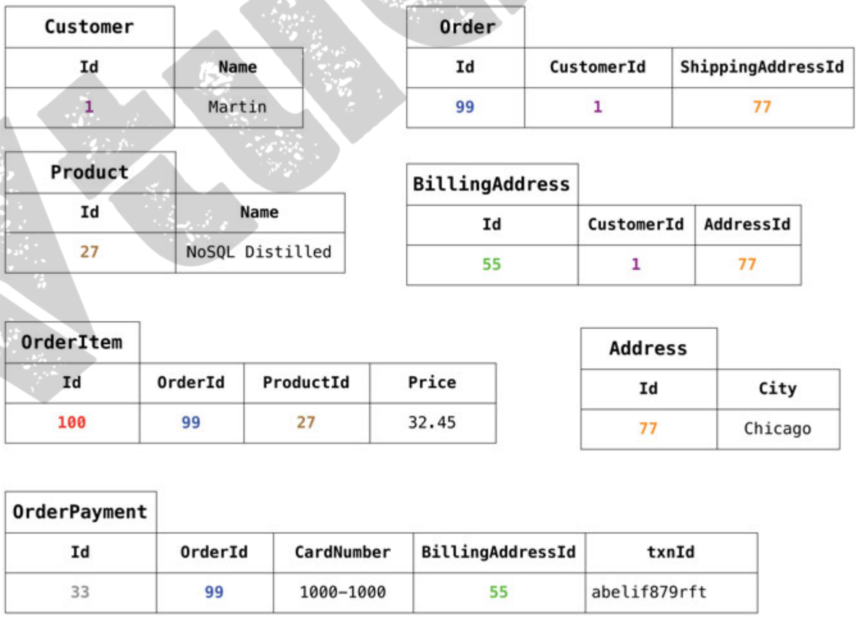


Figure 2.1 *Data model oriented around a relational database (using UML notation [Fowler UML])*



Customer

Id	Name
1	Martin

Order

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*

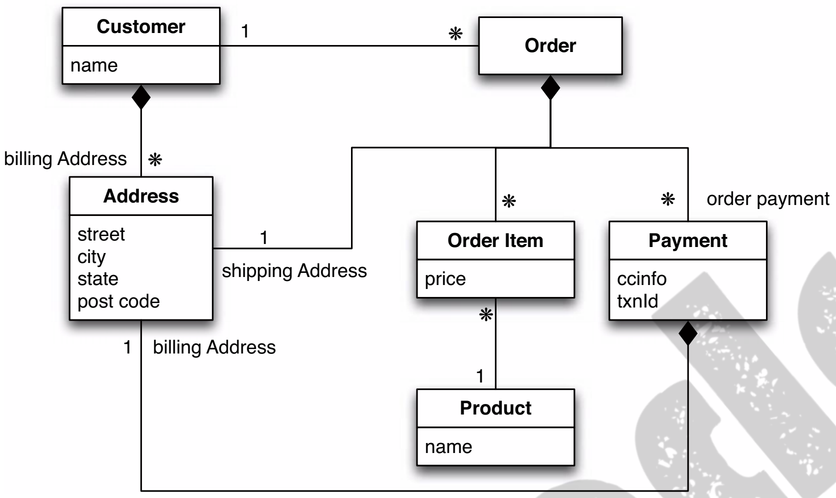


Figure 2.3 *An aggregate data model*

Again, we have some sample data, which we'll show in JSON format as that's a common representation for data in NoSQL land.

```
// 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).

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

```
// 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"}]
```

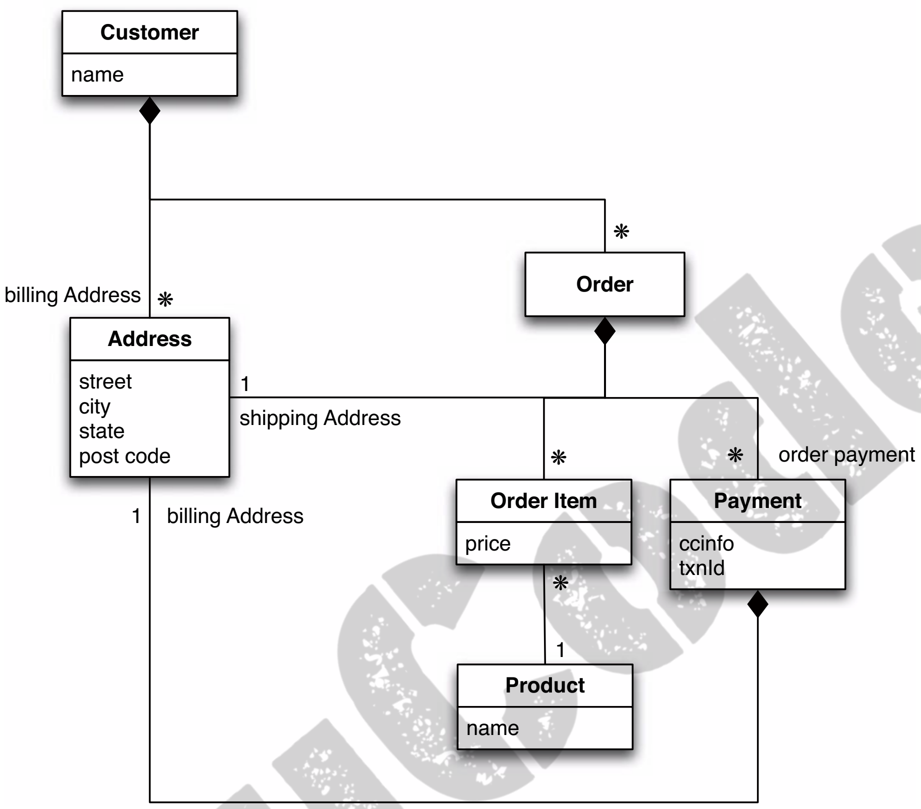


Figure 2.4 *Embed all the objects for customer and the customer's orders*

```
  "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 the 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')`.

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.

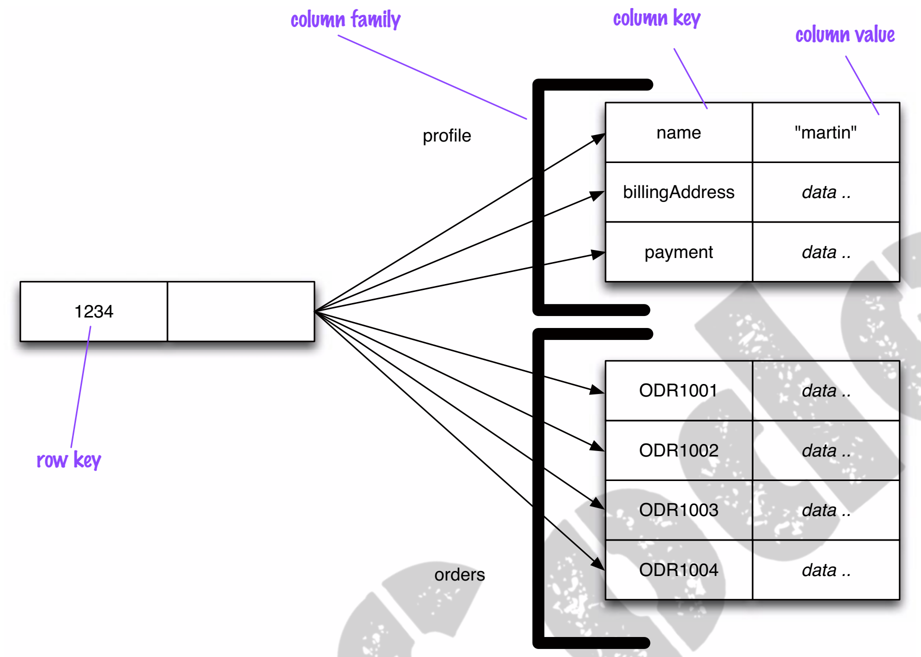


Figure 2.5 *Representing customer information in a column-family structure*

- 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.

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.

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

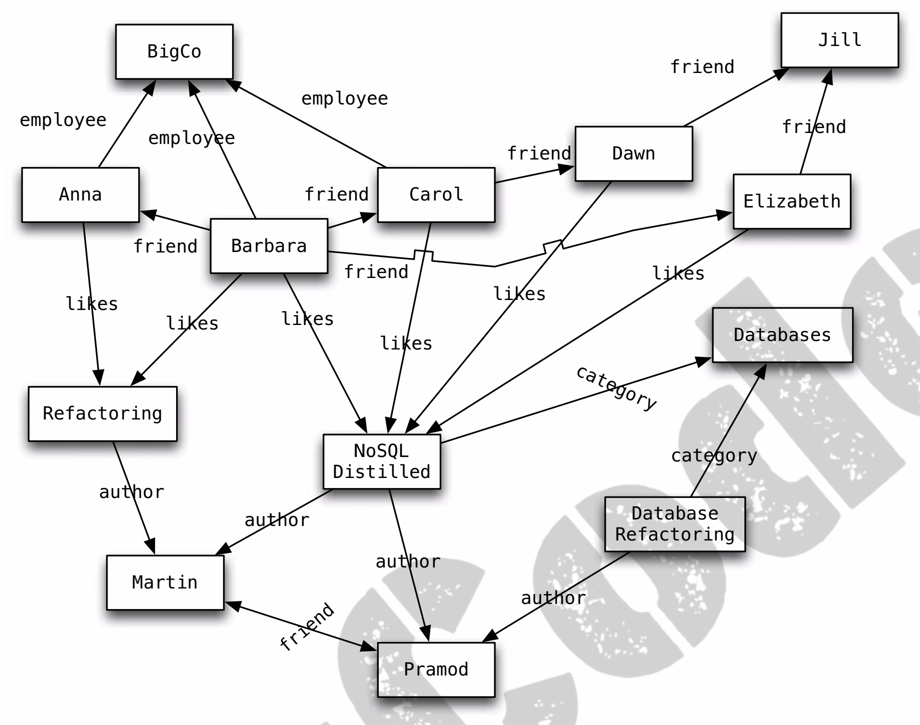


Figure 3.1 *An example graph structure*

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

```
//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 in 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).

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

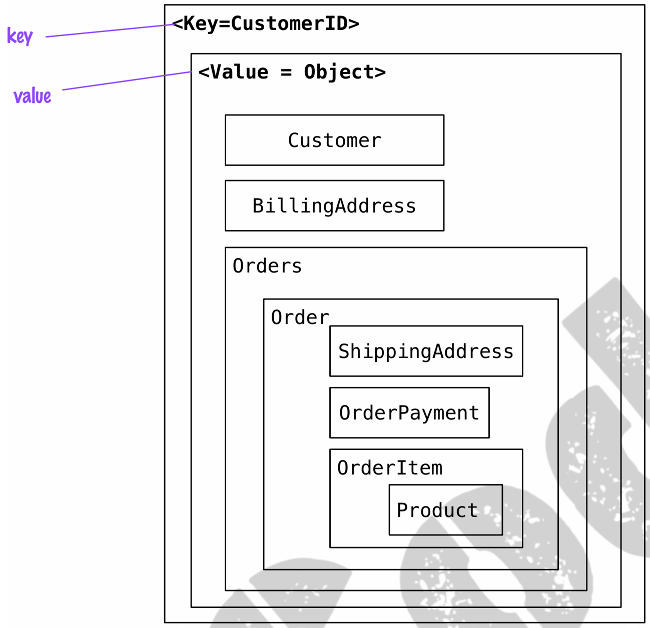


Figure 3.2 *Embed all the objects for customer and their orders.*

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`.

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

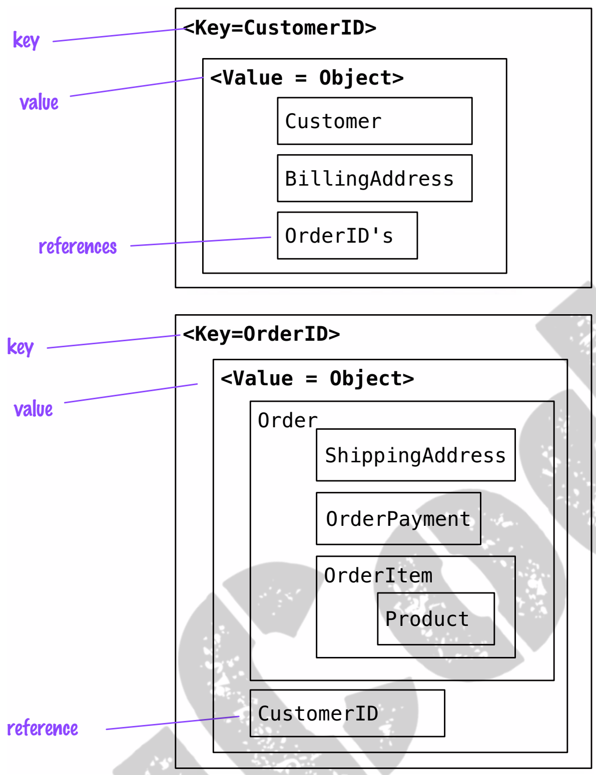


Figure 3.3 `Customer` *is stored separately from* `Order`.

```
# 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"}
  }
}
```

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.

```
{
"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`.

```
# 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.

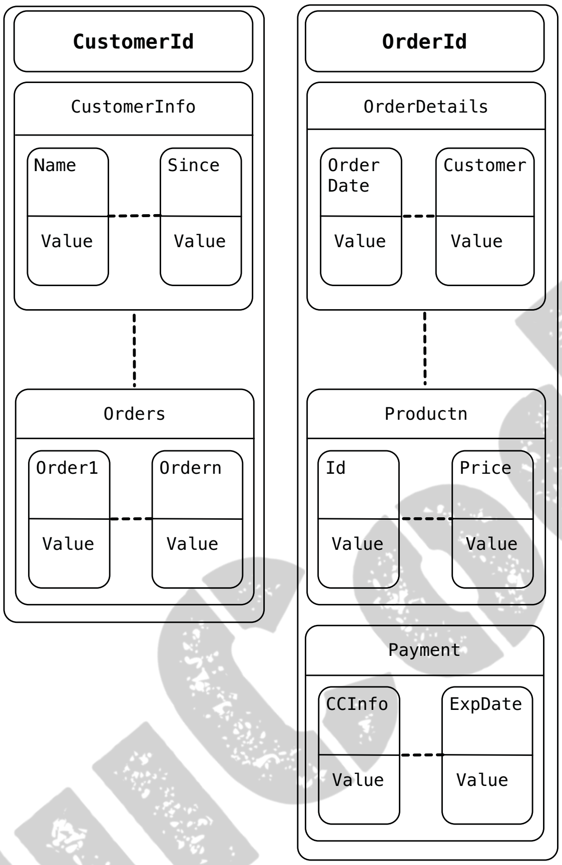


Figure 3.4 *Conceptual view into a column data store*

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.

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 Databases*. All we need to do is query for the product node `Refactoring Databases` and look for all the `Customers` with the incoming *PURCHASED* relationship.

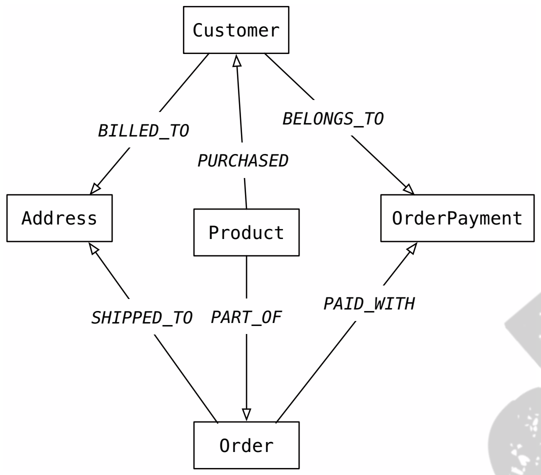


Figure 3.5 *Graph model of e-commerce data*

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.