# Unit-IV
# NoSQL Management

## Why NoSQL?

- 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.
- There have been times when adatabase 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.
- We will discuss
  - why relational databases became so dominant? and
  - why current rise of NoSQL databases isn't a flash in pan.

## The Value of Relational Databases

- Relational databases have become such an embedded part of our computing culture
- Benefits of Relational databases
  - Getting at Persistent Data
  - Concurrency
  - Integration
  - A (Mostly) Standard Model

## The Value of Relational Databases

### Getting at Persistent Data

- ❑ Probably the most obvious value of a database is keeping large amounts of persistent data.
- ❑ 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.
- ❑ To keep data around, write it to a backing store (disk)
- ❑ For most enterprise applications, however, backing store is a database.

## The Value of Relational Databases

### Concurrency

- Enterprise applications tend to have many people modifying same data.
- Need to coordinating these interactions (e.g. avoid double booking of hotel rooms)
- Concurrency is notoriously difficult to get right
- Enterprise applications - 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.
- Transactional mechanism has worked well to contain the complexity of concurrency.
- With transactions, make a change, and if an error occurs during processing of change then roll back transaction to clean things up.

## The Value of Relational Databases

### Integration

- ❑ Enterprise applications live in a rich ecosystem that requires multiple applications, written by different teams.
- ❑ 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 where multiple applications store their data in a single database.
- ❑ Using a single database allows all the applications to use each others’ data easily

## The Value of Relational Databases

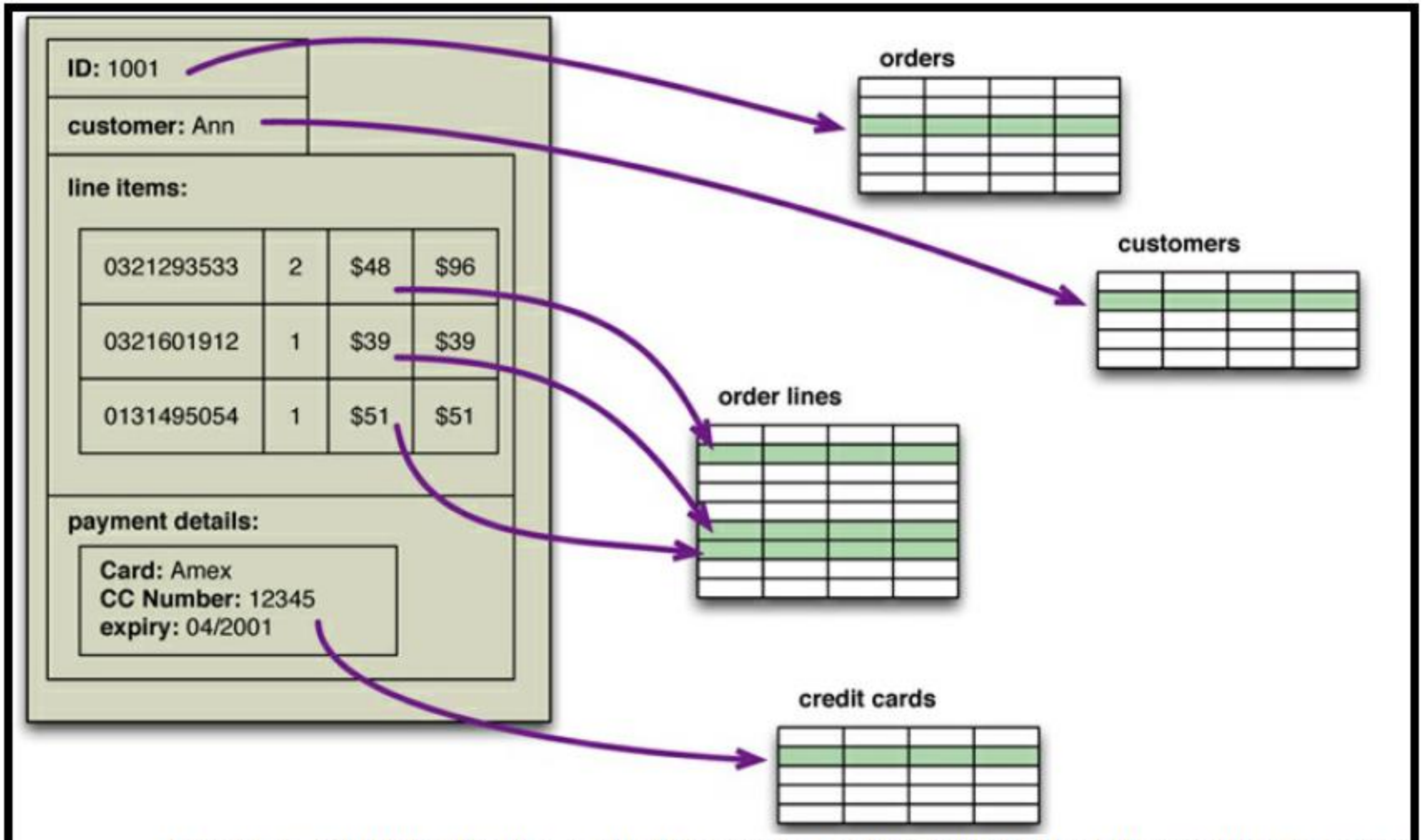
### Standard Model

- ❑ Relational databases have succeeded because they provide the core benefits in a standard way.
- ❑ Developers and database professionals can learn the basic relational model and apply it in many projects.
- ❑ Although there are differences between different relational databases, core mechanisms remain the same.
- ❑ Different vendors’ SQL dialects are similar, transactions operate in mostly the same way.

## Impedance Mismatch

- Relational databases provide many advantages, but they are by no means perfect
- Biggest frustration has been what's commonly called the **impedance mismatch:** difference between 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.
- All operations in SQL consume and return relations
- Limitations : 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, can take on much richer structures than relations.
- If 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.

# Impedance Mismatch



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

## Impedance Mismatch

- Impedance mismatch is a major source of frustration to application developers
- Object-oriented languages succeeded in becoming the major force in programming,
- Impedance mismatch has been made much easier to deal with by the wide availability of object relational mapping (ORM) frameworks, such as Hibernate and iBATIS that implement well-known mapping patterns
- Object-relational mapping frameworks remove a lot of grunt work
- Relational databases continued to dominate enterprise computing world in the 2000s, but during that decade cracks began to open in their dominance.

# Application and Integration Databases

- Primary factor for popularity of relational database was the role of SQL as an integration mechanism between applications
- Database acts as integration database—with multiple applications, usually developed by separate teams, storing their data in a common database.
- Downsides to shared database integration
  - A structure that's designed to integrate many applications ends up being more complex
  - data storage changes must coordinate with all other applications using the database
  - Different applications have different structural and performance needs
  - database usually cannot trust applications to update data in a way that preserves database integrity
- Different approach is treat your database as an **application database**

## Application and Integration Databases

- ❑ 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.

## Attack of the Clusters

- In 2000s did see several large web properties dramatically increase in scale.
- 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.

# Attack of the Clusters

- ❑ Scaling up implies bigger machines, more processors, disk storage, and memory.
- ❑ Scale out alternative is to use lots of small machines in a cluster.
- ❑ It can also be more resilient—while individual machine failures are common, overall cluster can be built to keep going despite such failures, providing high reliability.
- ❑ As large properties moved towards clusters
- ❑ 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
- ❑ Relational databases could also be run as separate servers for different sets of data
- ❑ Sharding has to be controlled by the application
- ❑ These technical issues are exacerbated by licensing costs

## Attack of the Clusters

- ❑ 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; and were capturing huge amounts of data.
- ❑ These things gave them the motive.
- ❑ 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.

# The Emergence of NoSQL

- 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.
- NoSQL databases don’t use SQL. Some of them do have query languages
- Cassandra’s CQL is like this—“exactly like SQL
- Most NoSQL databases are driven by the need to run on clusters
- This has an effect on their data model as well as their approach to consistency.
- Relational databases use ACID transactions to handle consistency across the whole database.
- NoSQL databases offer a range of options for consistency and distribution.

# The Emergence of NoSQL

- ❑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
- ❑Graph databases offers a different data model that makes it better at handling data with complex relationships.
- ❑NoSQL databases operate without a schema
- ❑NoSQL has opened up the range of options for data storage

## The Emergence of NoSQL

- 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
- 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 Emergence of NoSQL

- Relational databases are going away—going to be the most common form of database in use.
- Still relational databases are recommended.
- 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.

# The Emergence of NoSQL

- To make this polyglot world work, our view is that organizations also need to shift from integration databases to application databases.
- NoSQL database is used as an application database; generally consider NoSQL databases a good choice for integration databases.
- History of NoSQL development - concentrate on big data running on clusters.
- An equally important reason is the old frustration with the impedance mismatch problem.
- Big data concerns have created an opportunity to think freshly about their data storage needs
- There are two primary reasons for considering NoSQL.
    1. One is to handle data access with sizes and performance that demand a cluster;
    2. the other is to improve the productivity of application development by using a more convenient data interaction style.

# Summary – Relational and NoSQL Databases

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

# Aggregate Data Models

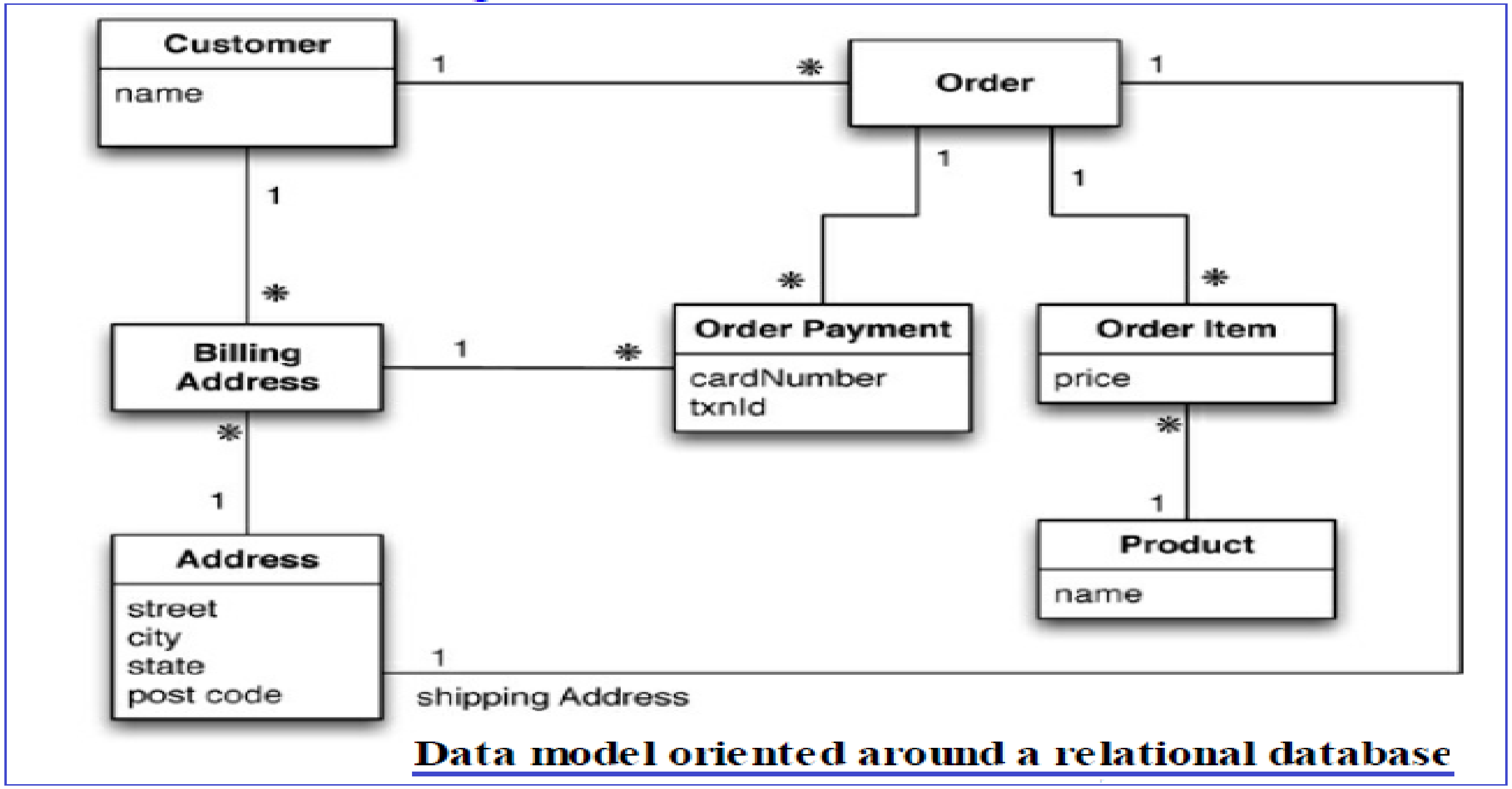
- A data model is the model through which we perceive and manipulate our data.
- Data model describes how to interact with the data in the database
- Storage model describes how the database stores and manipulates the data internally
- Term "data model" often means the model of the specific data in an application
- A developer might point to an entity-relationship diagram of their database and refer to that as their data model
- We will be using "data model" to refer to the model by which the database organizes data—what might be more formally called a metamodel.
- Dominant data model of the last couple of decades is relational data model.
- NoSQL solution has a four different model that it uses in the NoSQL ecosystem:
    1. key-value,
    2. document,
    3. column-family, and
    4. graph.
- First three share a common characteristic of their data models - **aggregate orientation**

# Aggregates

- Relational model takes information to be stored and divides it into tuples (rows).
- A tuple is a limited data structure: captures a set of values, cannot nest one tuple within another to get nested records
- Aggregate orientation approach - recognizes that often, you want to operate on data in units that have a more complex structure than a set of tuples.
- |There is no common term for this complex record; called as "aggregate."
- Aggregate is a term that comes from Domain-Driven Design.
- **In Domain-Driven Design, an aggregate is a collection of related objects that we wish to treat as a unit.**
- 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 databases to handle operating on a cluster
- Aggregate makes a natural unit for replication and sharding.
- Application programmers often manipulate data through aggregate structures

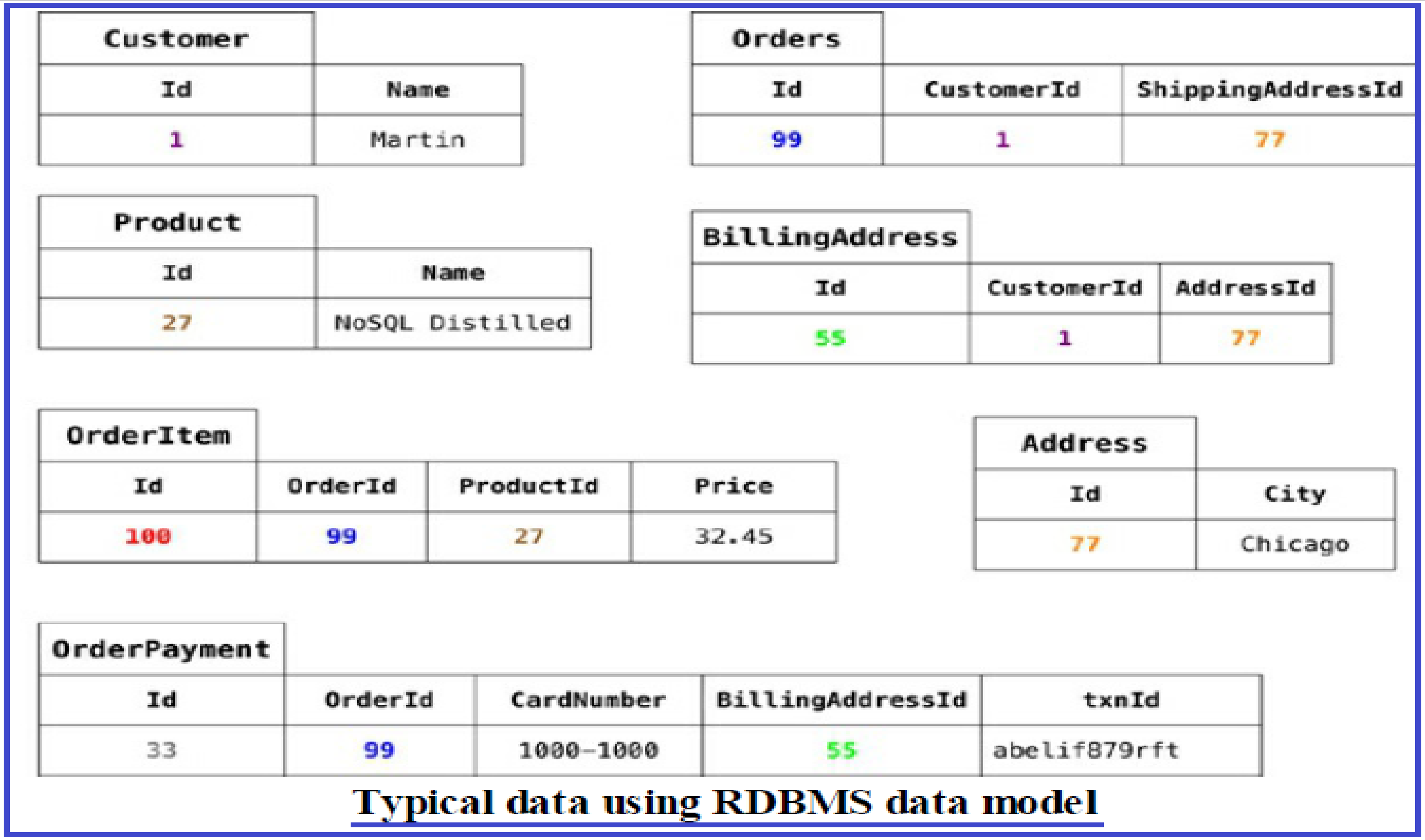
## Example of Relations and Aggregates

- For e-commerce website want to store information about users, our product catalog, orders, shipping addresses, billing addresses, and payment data.
- Can use this scenario to model data using a relation data store as well as NoSQL data stores and talk about their pros and cons



Data model oriented around a relational database

# Example of Relations and Aggregates



**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**

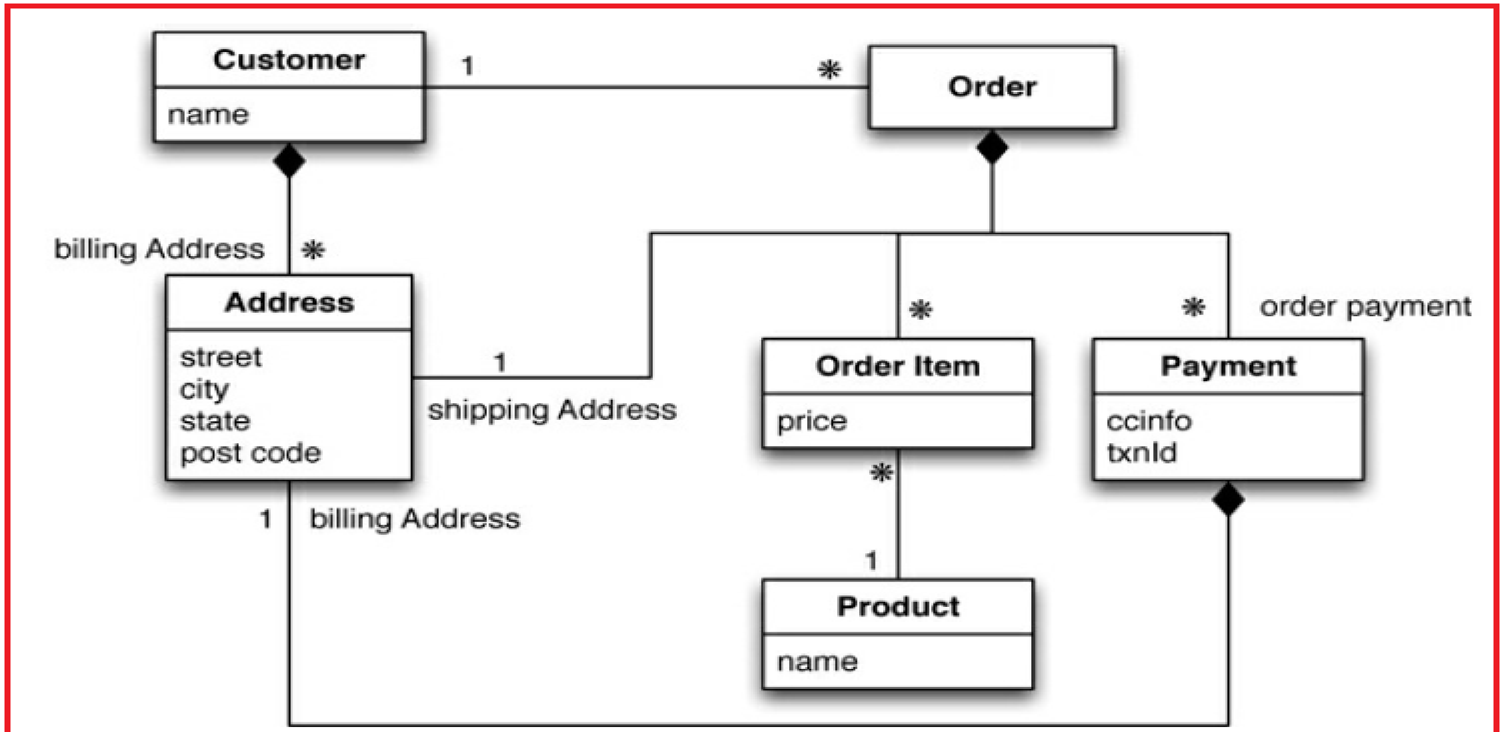
| Id | City |
|---|---|
| 77 | Chicago |

**OrderPayment**

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

Typical data using RDBMS data model

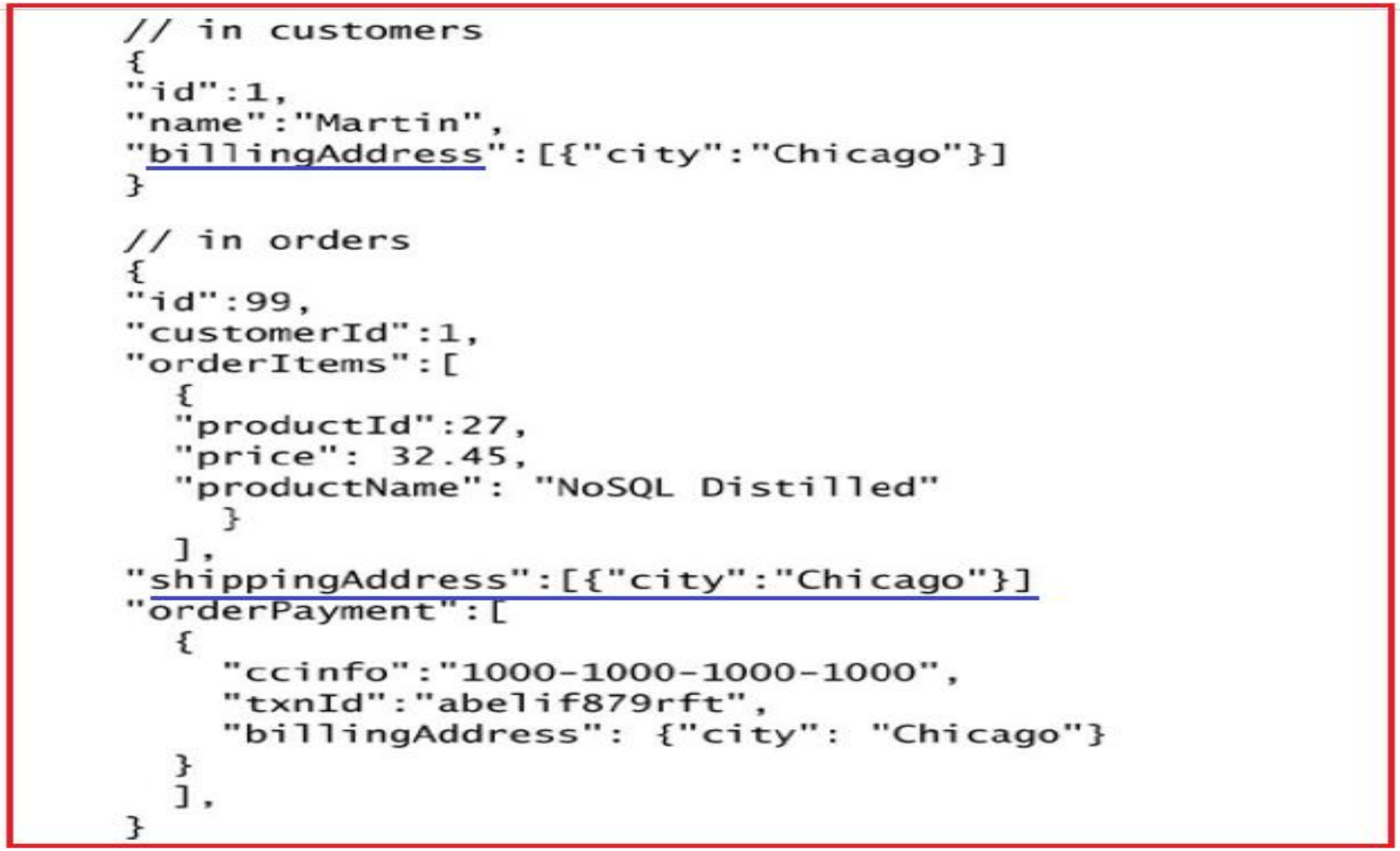
# Example of Relations and Aggregates



An aggregate data model

- In this model, we have two main aggregates: **customer and order**
- 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.**

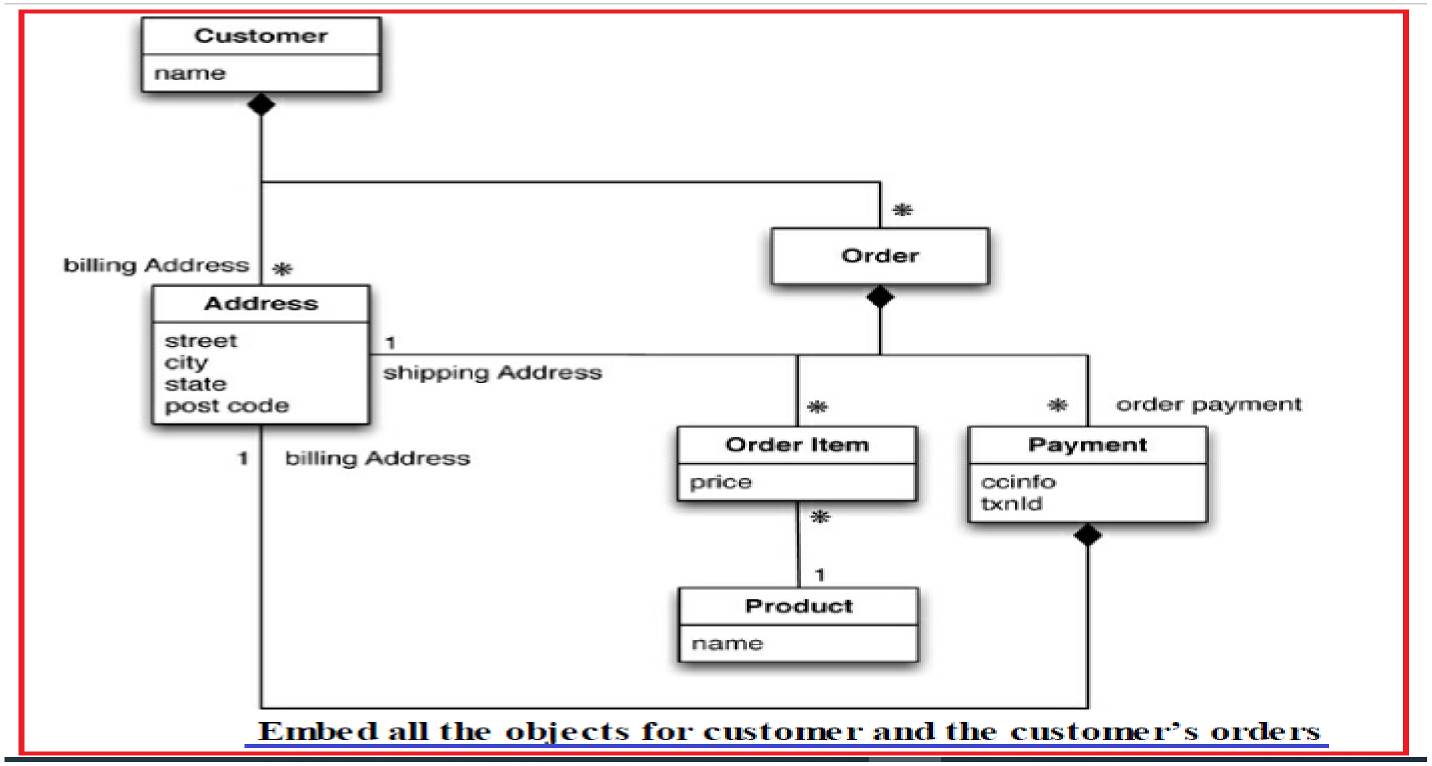
# Example of Relations and Aggregates



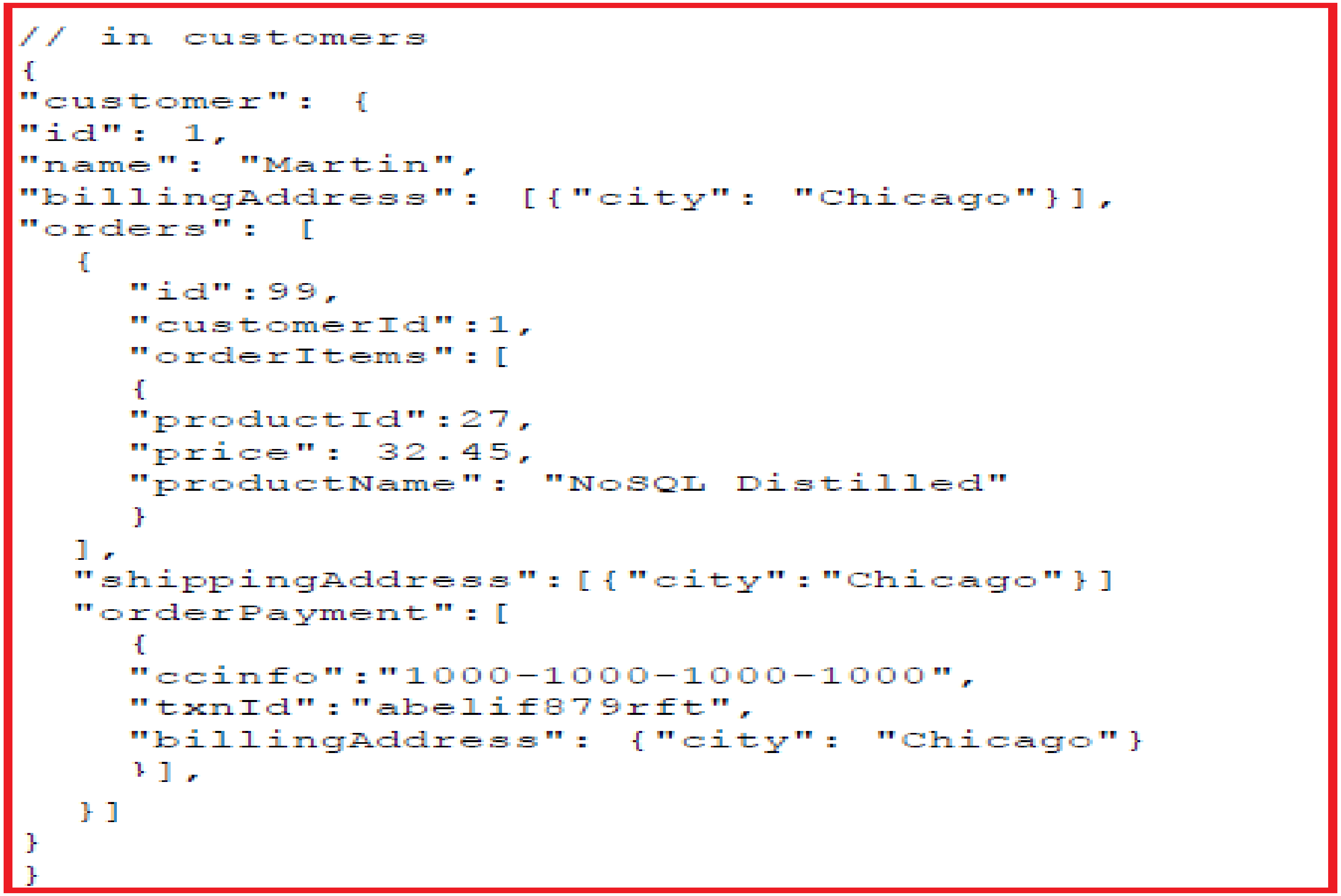
```
// in customers
{
"id":1,
"name":"Martin",
"billingAddress":[{"city":"Chicago"}]
}

// 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"}
  }
  ],
}
```

# Example of Relations and Aggregates

- Have to think about accessing that data—and make that part of 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.



# Example of Relations and Aggregates



```
// 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"}
    }],
  }]
}
}
```

## Example of Relations and Aggregates

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

## Consequences of Aggregate Orientation

- ❑ While the relational mapping captures the various data elements and their relationships 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.
- ❑ When working with aggregate-oriented databases, clearer semantics is to consider by focusing on the unit of interaction with data storage.
- ❑ Not a logical data property: It’s all about how the data is being used by applications.

# Consequences of Aggregate Orientation

- 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

## Consequences of Aggregate Orientation

- The clinching reason for aggregate orientation is that it helps greatly with running on a cluster.
- 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 ACID transactions.
- NoSQL databases don't support ACID transactions and thus sacrifice consistency.
- 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.
- Graph and other aggregate-ignorant databases usually do support ACID transactions

## Key-Value and Document Data Models

- ❑ Key-value and document databases were strongly aggregate-oriented.
- ❑ 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
- ❑ 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.

## Key-Value and Document Data Models

- In practice, 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 aggregate into lists or sets.
- Riak 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, 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 internal structure of document; this might be a key, but it's more likely to be something else.

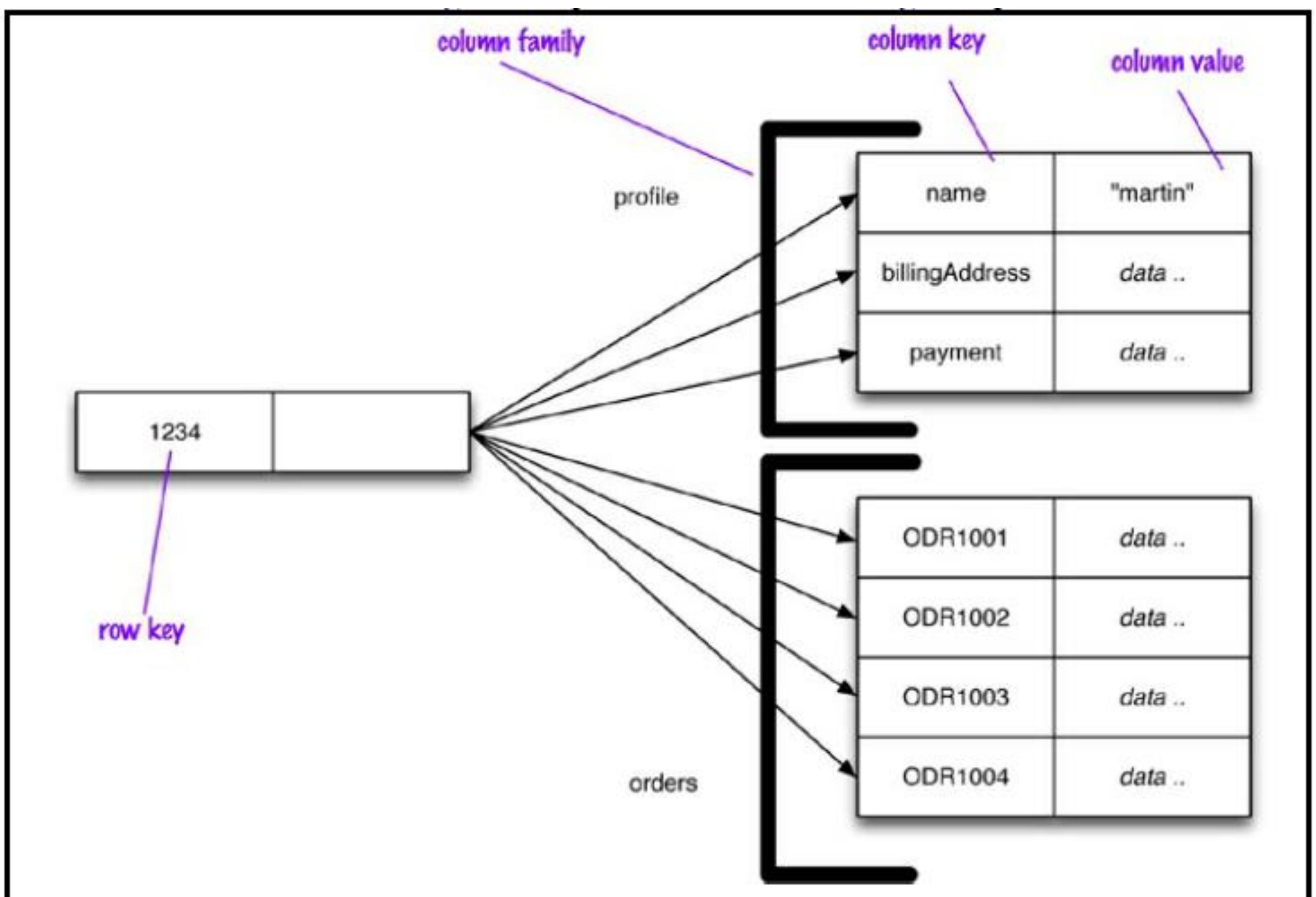
## Column-Family Stores

- ❑ Early and influential NoSQL databases was Google’s BigTable
- ❑ Its name conjured up a tabular structure which it realized with sparse columns and no schema.
- ❑ 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.
- ❑ Bigtable-style data model are often referred to as column stores
- ❑ Most databases have a row as a unit of storage which, helps write performance.
- ❑ When 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 basic storage unit—which is why these databases are called column stores.

## Column-Family Stores

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

# Column-Family Stores



Representing customer information in a column-family structure

# Column-Family Stores

This also gives you a couple of ways to think about how 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.
- Document database declares some structure to the database, each document is still seen as a single unit.
- Column families give a two-dimensional quality

## Aggregation - 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.

## More Details on Data Models

- Discussed 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.

## Relationships

- Aggregates are useful - 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 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.

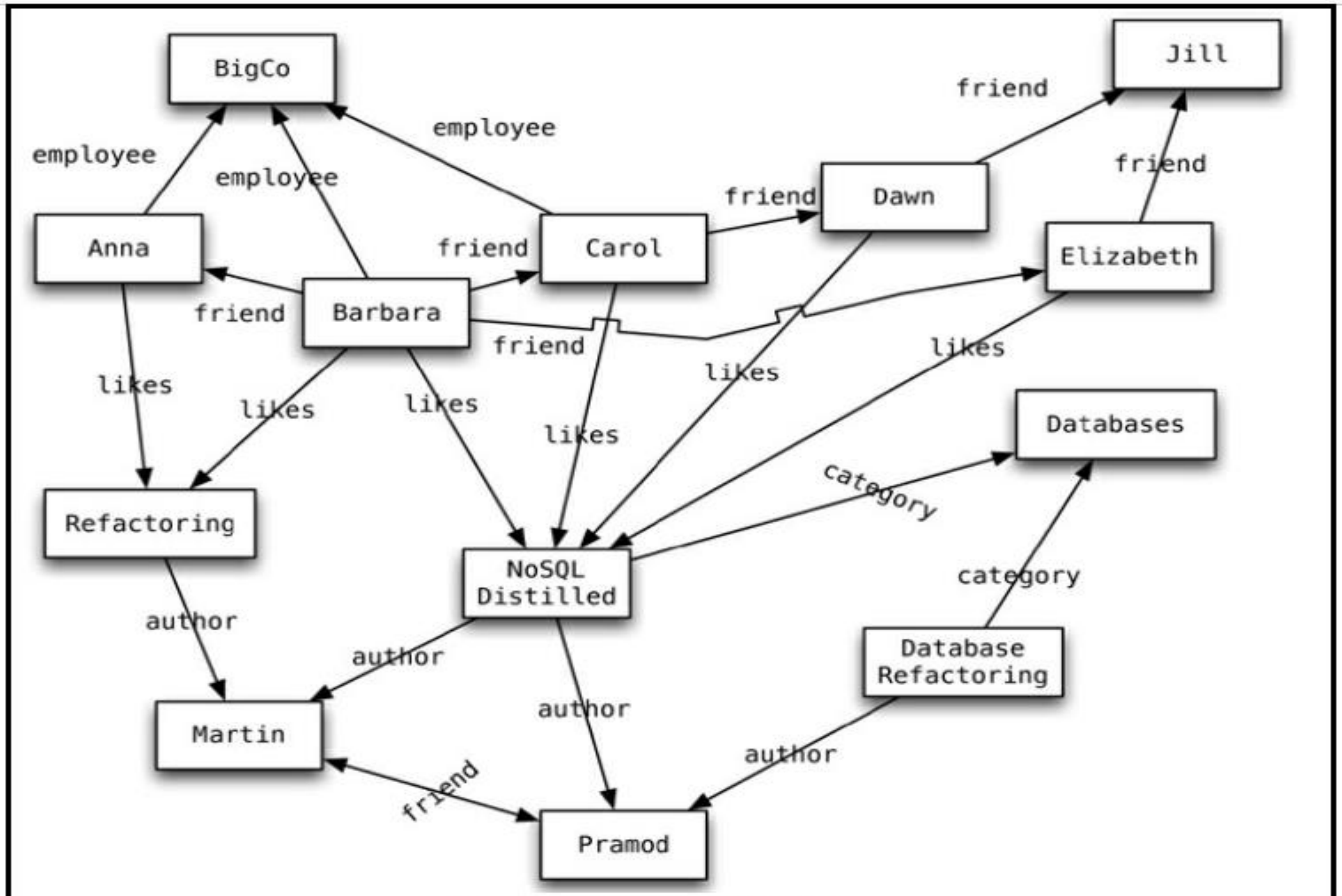
# Relationships

- ❑ 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.
- ❑ 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 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 to modify multiple records in a single transaction, providing ACID guarantees while altering many rows.
- ❑ Aggregate-oriented databases become more awkward as you need to operate across multiple aggregates.

## 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.

# Graph Databases



An example graph structure

# Graph Databases

- ❑ Fundamental data model of a graph database is very simple: nodes connected by edges (also called arcs).
- ❑ Variation in data models— what mechanisms you have to store data in your nodes and edges.
- ❑ 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 schema less fashion
- ❑ Infinite Graph stores your Java objects, which are subclasses of its built-in types, as nodes and edges.
- ❑ Graph database allows to query network with query operations designed
- ❑ Relational database – complex relationships – expensive poor performance

# Graph Databases

- Graph databases make traversal along the relationships very cheap.
- Data is found by navigating through the network of edges
- Emphasis on relationships makes graph databases very different from aggregate-oriented databases
- Graph databases run on a single server rather than distributed across clusters
- ACID transactions need to cover multiple nodes and edges to maintain consistency.
- Common thing of Graph databases with aggregate-oriented databases is -
  - their rejection of the relational model and
  - an upsurge in attention received around same time as the rest of the NoSQL field.

# Schemaless Databases

- ❑ Common theme across of NoSQL databases is that they are schema less.
- ❑ Relational database - first have to define a schema—the store data.
- ❑ Before data storage , schema have to defined schema
- ❑ With NoSQL databases, storing data is much more casual.
- ❑ A key-value store allows you to store any data you like under a key.
- ❑ Document database - no restrictions on the structure of the documents to be stored.
- ❑ Column-family databases allow you to store any data under any column.
- ❑ Graph databases allows to freely add new edges and freely add properties to nodes and edges as you wish.
- ❑ With a schema, have to figure out in advance what you need to store
- ❑ Can easily change data storage as learned more about your project.

## Schemaless Databases

- 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 is 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.
- Schemalessness avoids this, allowing each record to contain just what it needs—no more, no less.
- Schemalessness avoids many problems that exist with fixed-schema databases.
- Schema is useful when displaying data in a report.

# Schemaless Databases

- 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)
   }
}
```

- Code assumes that certain field names are present and carry data with a certain meaning
- In 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.

# Schemaless Databases

- Having the implicit schema in the application code results in some problems.
- In order to understand what data is present you have to dig into the application code.
- Well structured code - able to find a clear place from which to deduce the schema. But there are no guarantees
- 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 validations upon that data to ensure that different applications don't manipulate data in an inconsistent way.
- These are reasons why relational databases have a fixed schema,
- Schemas have value, and the rejection of schemas by NoSQL databases is indeed quite startling.

# Schemaless Databases

- Schemaless database shifts the schema into the application code
- 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.
  - 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.
- Nonuniformity in data is a good reason to favor a schemaless database.

# Materialized Views

- Discussed aggregate-oriented data models, their advantages.
- Want to access orders, have to store all the data for an order contained in a single aggregate
- 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.
- Can reduce this burden by building an index on the product, but you're still working against the aggregate structure.
- Relational databases - lack of aggregate structure and can access data in different ways.
- Convenient mechanism in relaxational database – views
- Views  allows to look at data differently from the way it's stored.
- 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 view—a handy form of encapsulation.

# Materialized Views

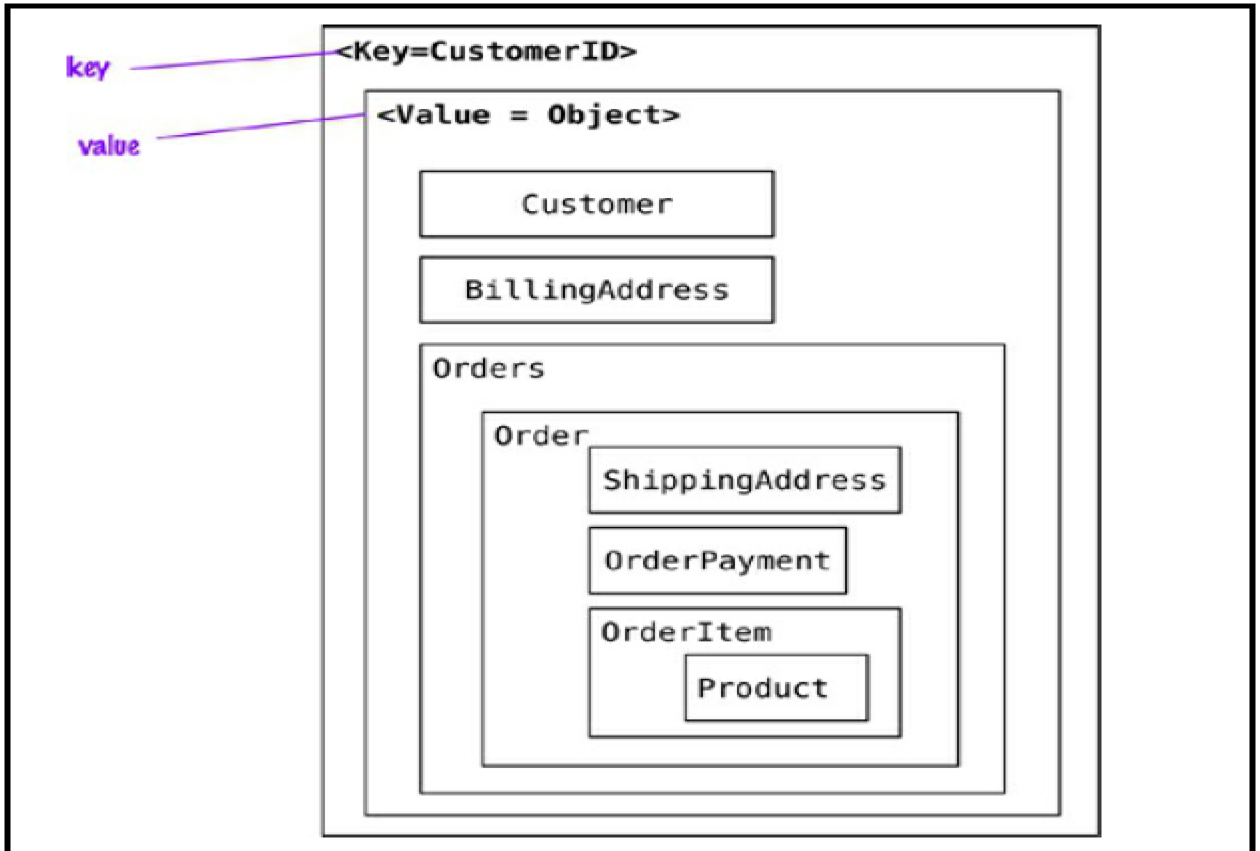
- ❑ Views provide a mechanism to hide from the client whether data is derived data or base data—but expensive to compute.
- ❑ To cope with this, **materialized views** were invented, which are views that are computed in advance and cached on disk.
- ❑ Materialized views are effective for data that is read heavily but can stand being somewhat stale (lacking freshness).
- ❑ NoSQL databases don’t have views, they may have precomputed and cached queries, and they reuse the term “materialized view” to describe them.
- ❑ Most applications have to deal with some queries that don’t fit well with aggregate structure
- ❑ NoSQL databases create materialized views using a map-reduce computation.

# Materialized Views

- Two rough strategies for 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 approach is valuable here as it makes it easier to ensure that any updates to base data also update materialized views.
- Can run batch jobs to update the materialized views at regular intervals.
- Can build materialized views outside of the database by reading the data, computing the view, and saving it back to the database.

# Modeling for Data Access

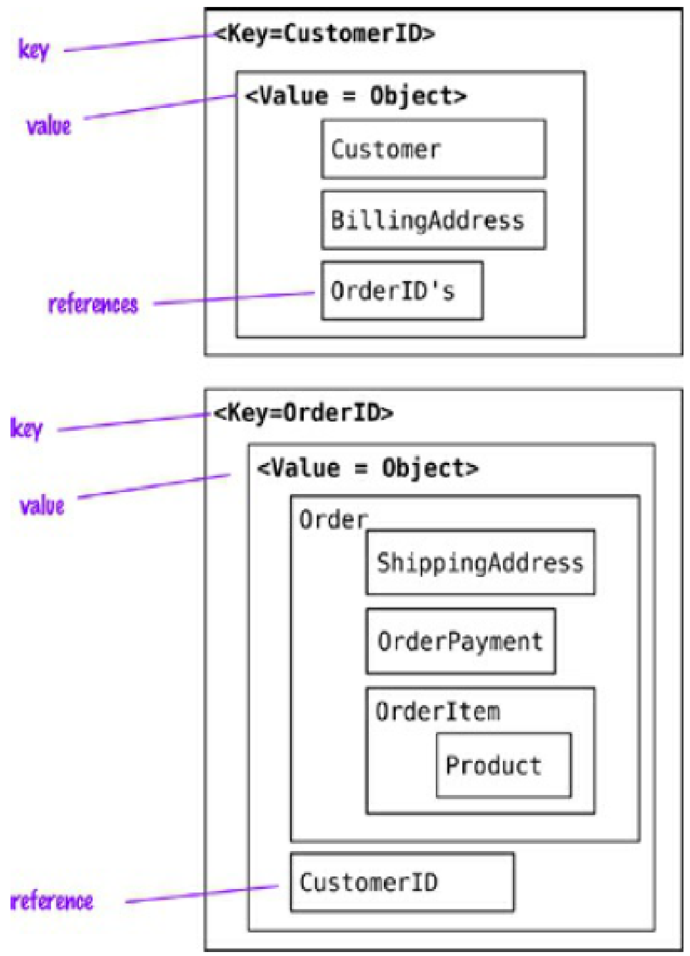
- When modeling data aggregates need to consider how the data is going to be read as well as what are the side effects on data related to those aggregates.
- Let's start with model where all data for the customer is embedded using a key-value store



Embed all the objects for customer and their orders.

# Modeling for Data Access

- ❑ Can split the value object into `Customer` and `Order` objects and then maintain these objects' references to each other.



```
# 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"}
   }
}
```

Customer is stored separately from Order.

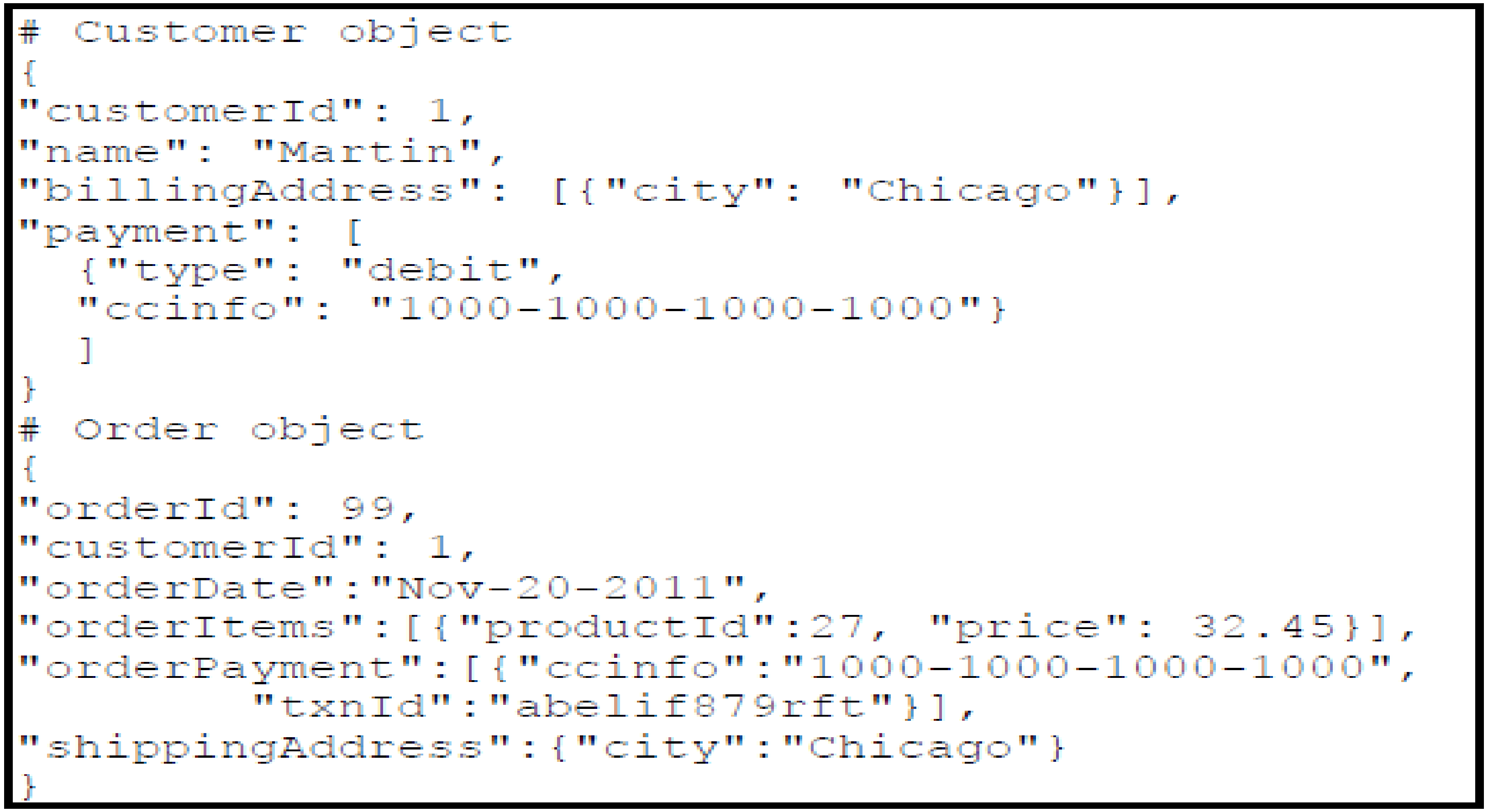
# Modeling for Data Access

- 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 and is the basis for **Real Time BI** or **Real Time Analytics**
- Enterprises don’t have to rely on end-of-the-day batch runs to populate data warehouse tables and generate analytics.

```
{  "
itemid":27,
"orders":{99,545,897,678}
}
{  "
itemid":29,
"orders":{199,545,704,819}
}
```

# Modeling for Data Access

- ❑ In document stores, we can query inside documents, can remove references to `Order`s from `Customer` object is possible.
- ❑ This change allows us to not update `Customer` object when new orders are placed.

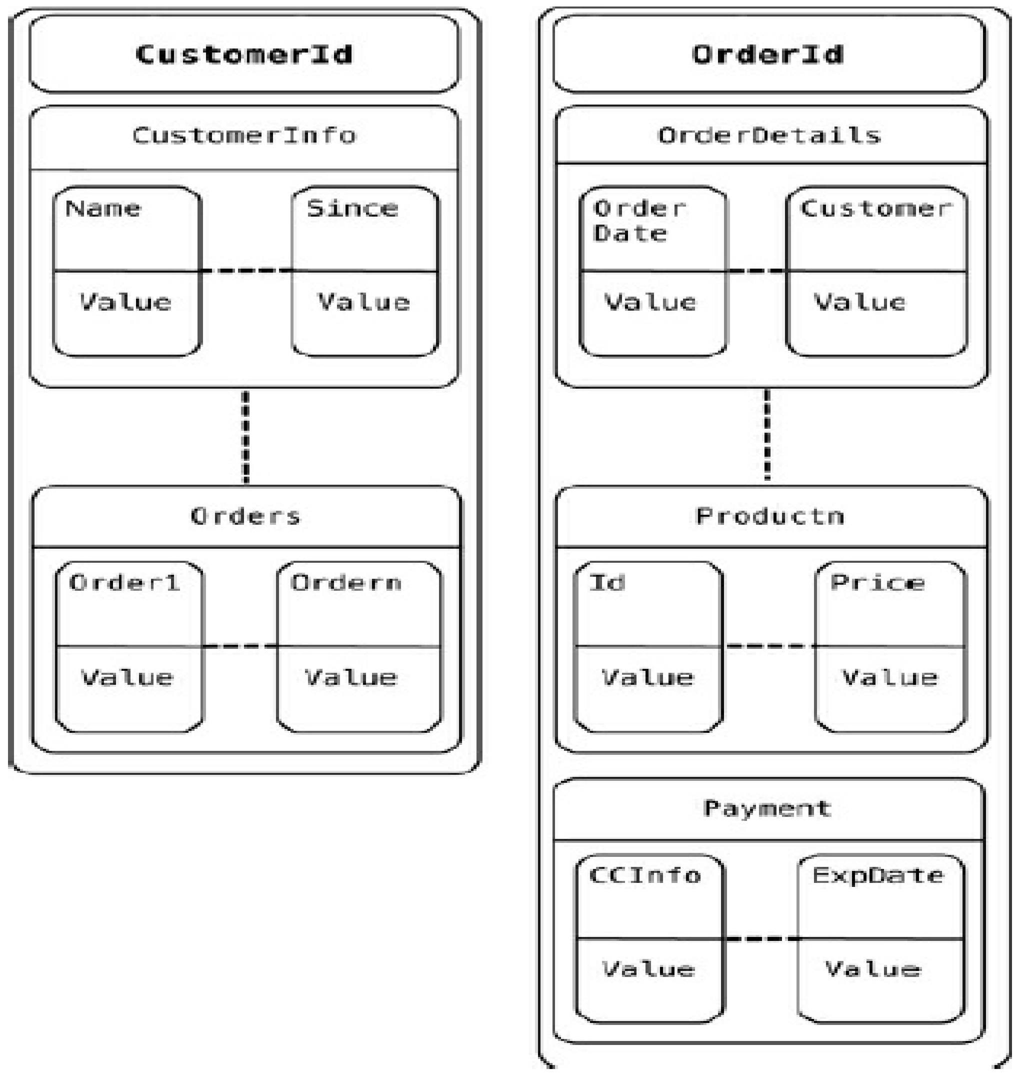


```
# Customer object
{
"customerId": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"payment": [
  {"type": "debit",
  "ccinfo": "1000-1000-1000-1000"}
  ]
}
# Order object
{
"orderId": 99,
"customerId": 1,
"orderDate":"Nov-20-2011",
"orderItems":[{"productId":27, "price": 32.45}],
"orderPayment":[{"ccinfo":"1000-1000-1000-1000",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

## Modeling for Data Access

- When modeling for column-family stores, 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.
- General rule is to make it easy to query and denormalize the data during write.

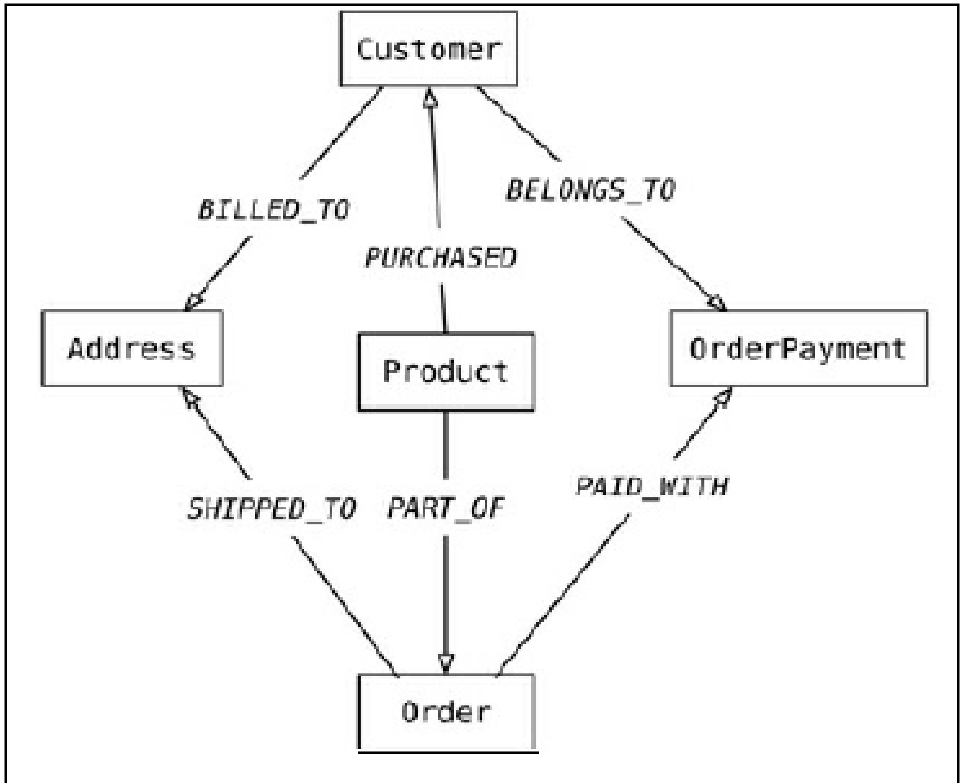
# Modeling for Data Access



Conceptual view into a column data store

# Modeling for Data Access

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



Graph model of e-commerce data

## Data Modelling – 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.

## Distribution Models

- 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, can 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.

## Distribution Models

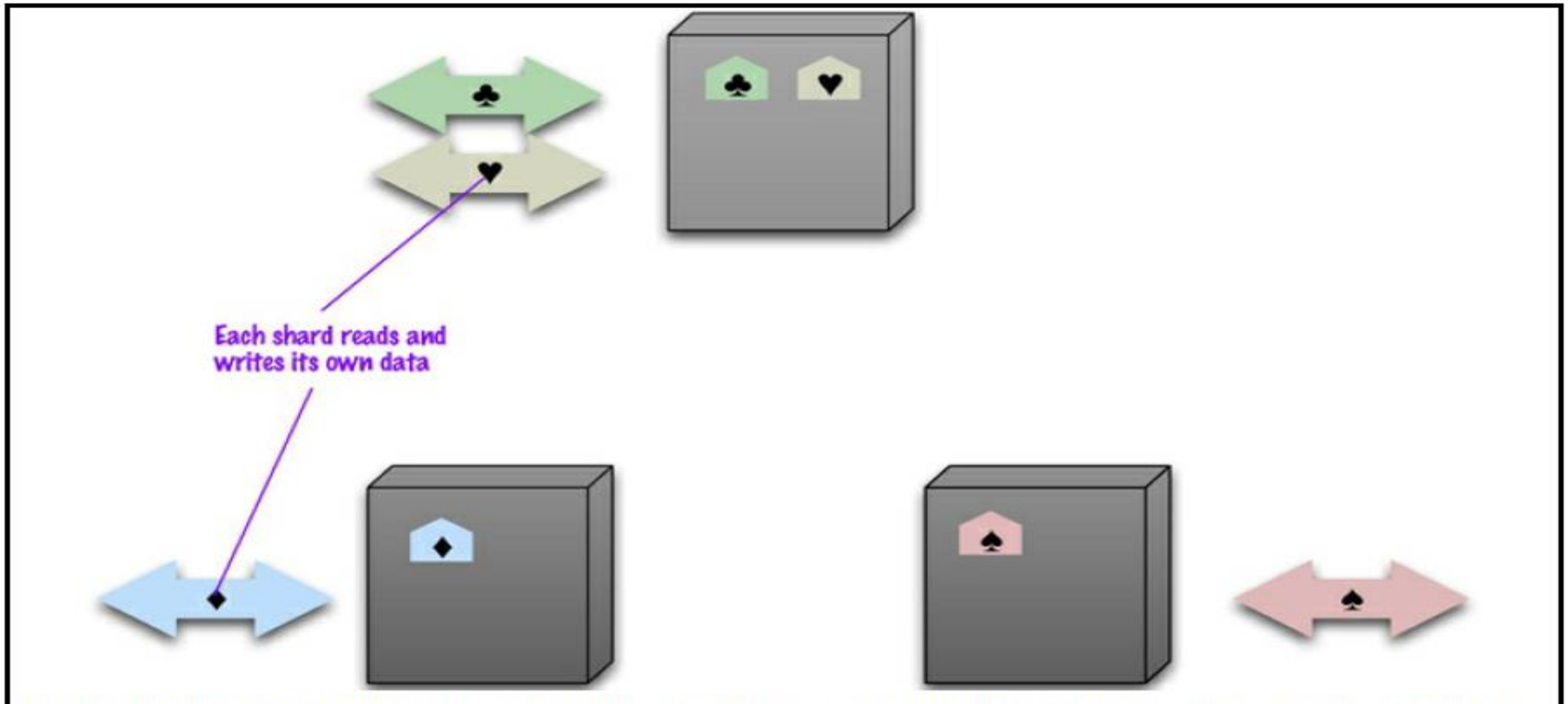
- Broadly, there are two paths to data distribution:
  - replication and
  - sharding.
- Replication takes the same data and copies it over multiple nodes. Sharding puts different data on different nodes.
- Replication and sharding are orthogonal techniques: You can use either or both of them.
- Replication comes into two forms:
  1. master-slave and
  2. peer-to-peer.
- Techniques :
  - first single-server,
  - master-slave replication,
  - sharding, and
  - finally peer-to-peer replication.

## Single Server

- ❑ Simplest distribution — no distribution at all.
- ❑ Run database on a single machine - the reads and writes to the data store.
- ❑ Eliminates all the complexities
- ❑ NoSQL databases are designed around the idea of running on a cluster
- ❑ Graph databases are the obvious category here—these work best in a single-server configuration.
- ❑ If data usage is processing aggregates, then a single-server document or key-value store may well be worthwhile
- ❑ If we can get away without distributing our data, we will always choose a single-server approach.

## Sharding

- Busy data store is busy - different people are accessing different parts of the dataset.
- Can support horizontal scalability by putting different parts of the data onto different servers—a technique that's called **sharding.**



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

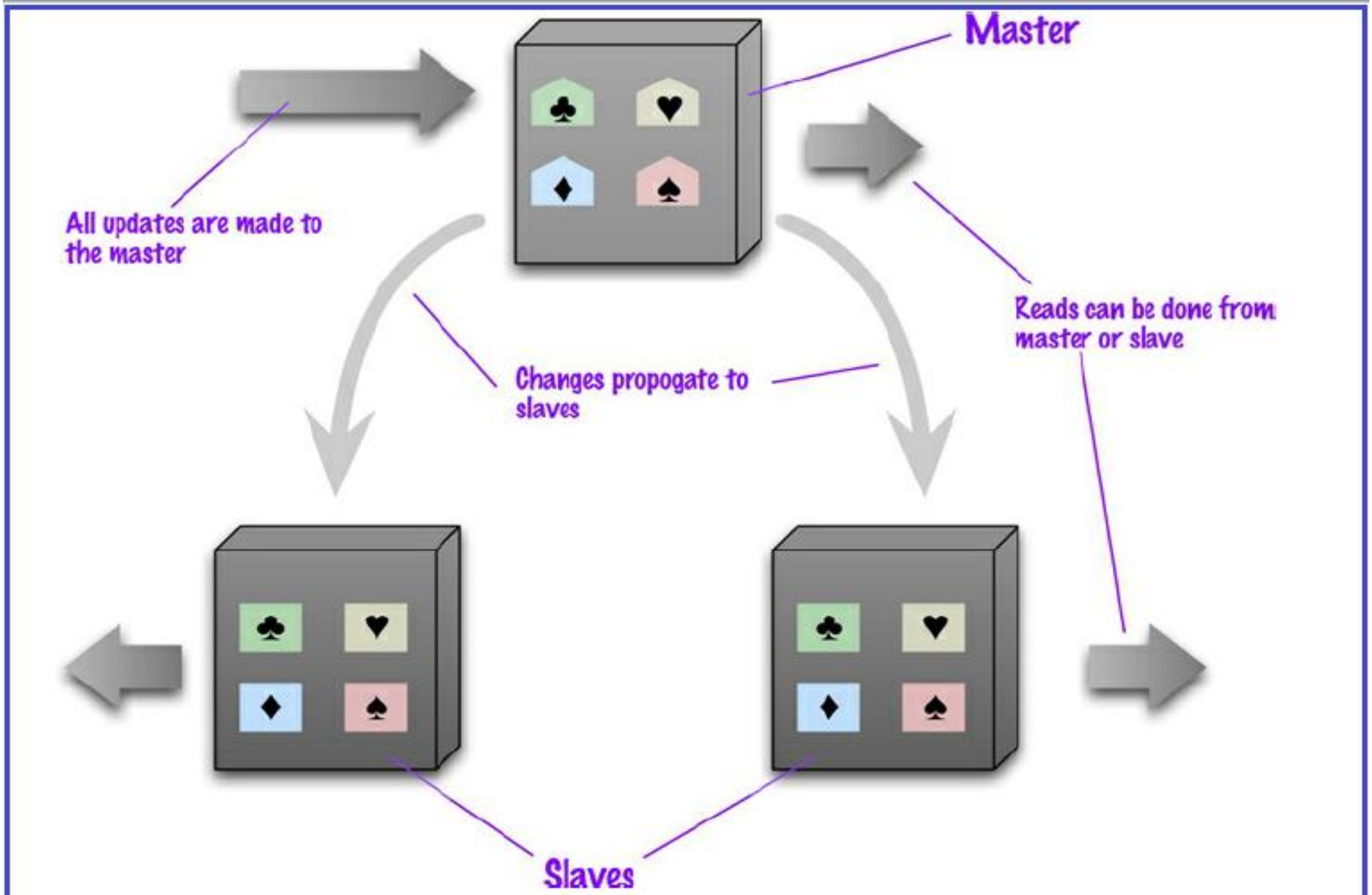
## Sharding

- Different users all talking to different server nodes
- How w 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
- Try to arrange aggregates so they are evenly distributed across the nodes which all get equal amounts of the load
- it’s useful to put aggregates together if you think they may be read in sequence
- Historically most people have done sharding as part of application logic
- 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
- Sharding does little to improve resilience when used alone

## Master-Slave Replication

- ❑With master-slave distribution, replicate data across multiple nodes.
- ❑One node is designated as the master, or primary.
- ❑ 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.
- ❑Master-slave replication is most helpful for scaling when you have a read-intensive dataset.
- ❑A second advantage of master-slave replication is read resilience: Should the master fail, the slaves can still handle read requests.

# Master-Slave Replication


Master
All updates are made to the master
Reads can be done from master or slave
Changes propogate to slaves
Slaves

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

# Master-Slave Replication

- ❑ Speed up recovery after a failure of the master - a slave can be appointed a new master very quickly.
- ❑ All read and write traffic can go to the master while the slave acts as a hot backup.
- ❑ Think of the system as a single-server store with a hot backup.
- ❑ 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.

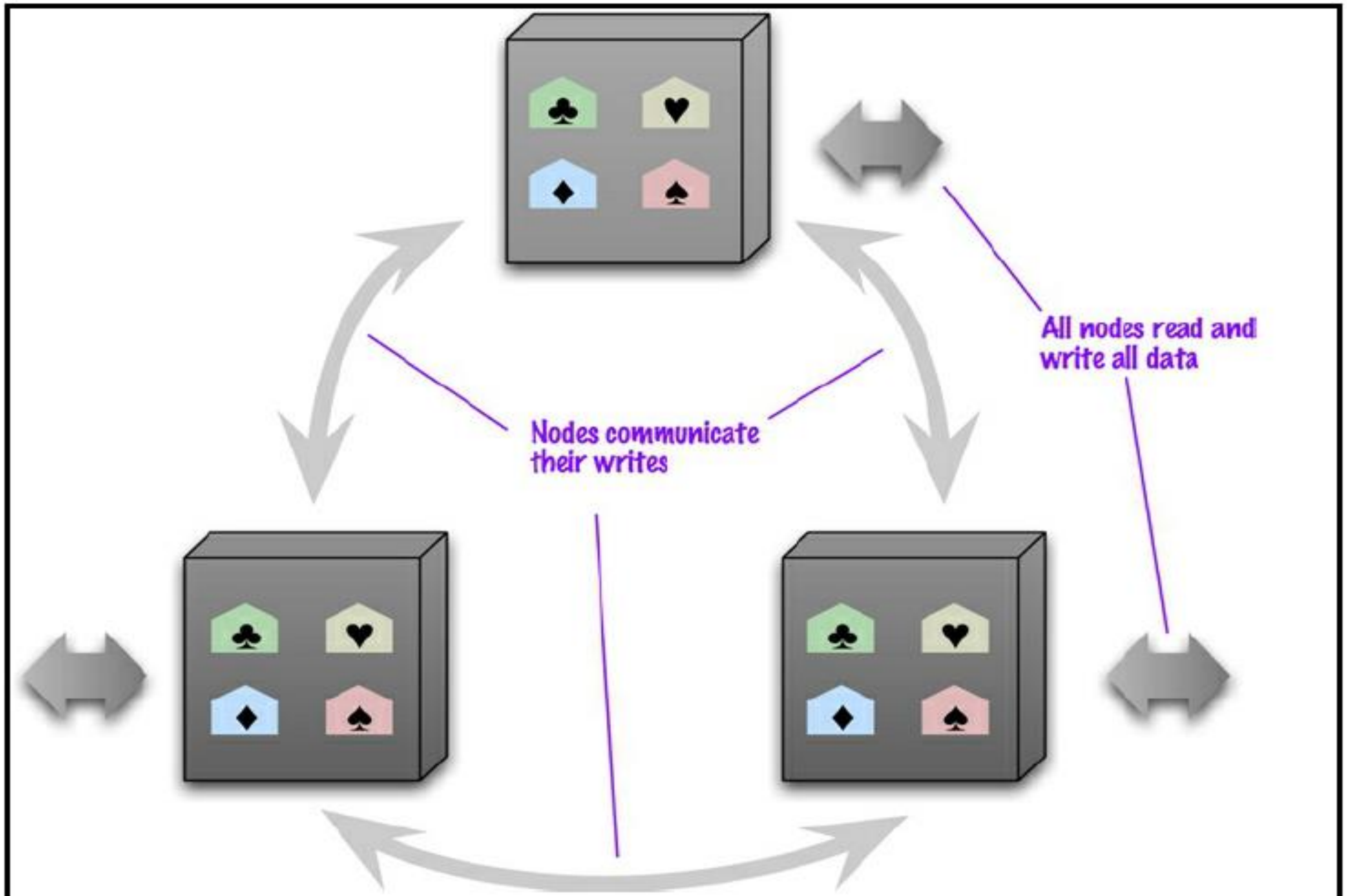
## Master-Slave Replication

- Replication comes with some alluring benefits, but it also comes with an inevitable dark side— inconsistency.
- 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.

## 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 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.
- ❑ Can ride over node failures without losing access to data
- ❑ Can easily add nodes to improve your performance
- ❑ The biggest complication is, again, consistency
- ❑ Two people will attempt to update the same record at same time—a write-write conflict

# Peer-to-Peer Replication

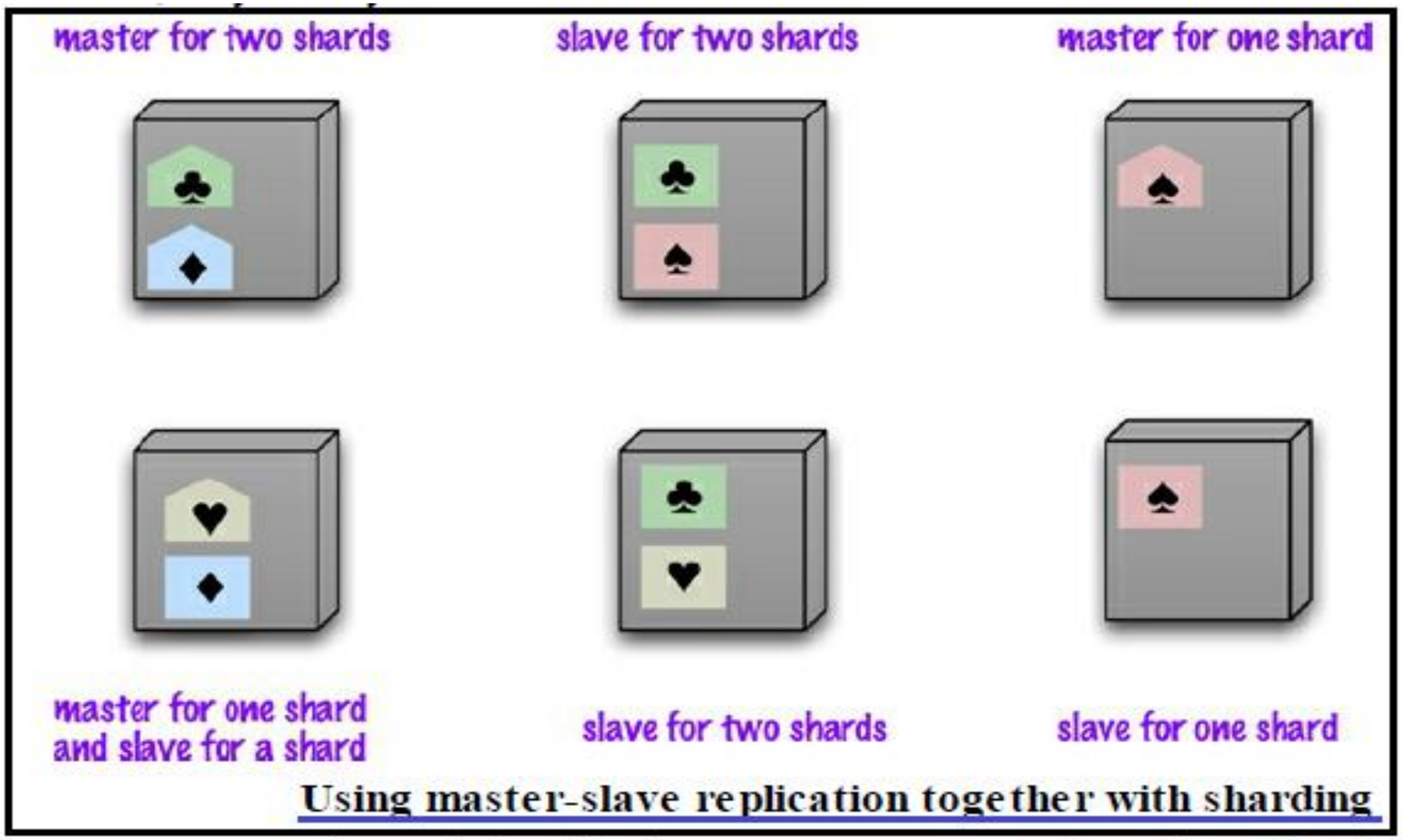

All nodes read and write all data
Nodes communicate their writes

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

## Combining Sharding and Replication

- ❑ Replication and sharding are strategies that can be combined.
- ❑ If we use both master-slave replication and sharding, 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.

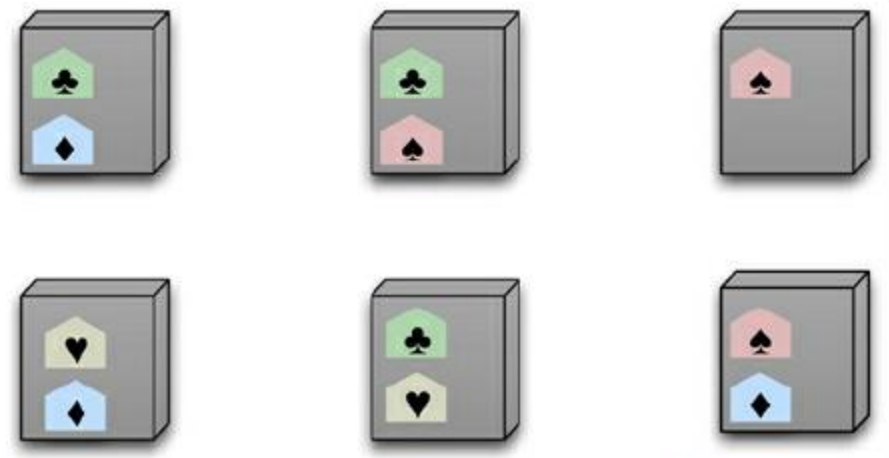
# Combining Sharding and Replication



Using master-slave replication together with sharding

# Combining Sharding and Replication

- 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



Using peer-to-peer replication together with sharding

## Distribution Models – 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

## Consistency

- ❑ 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**
- ❑ Consistency comes in various covers a myriad of ways errors
- ❑ Discuss about the various shapes consistency can take
  - ➢ Update Consistency
  - ➢ Read Consistency

## Update Consistency

- Consider updating a telephone number.
- Martin and Pramod 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
- In this case Martin’s is a lost update
- 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.

# Update Consistency

- 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
- 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.
- Both the pessimistic and optimistic approaches rely on a consistent serialization of the updates
- When people talk about concurrency in distributed systems, they talk about sequential consistency—ensuring that all nodes apply operations in the same order.
- Another optimistic way to handle a write-write conflict—save both updates and **record that they are in conflict**.

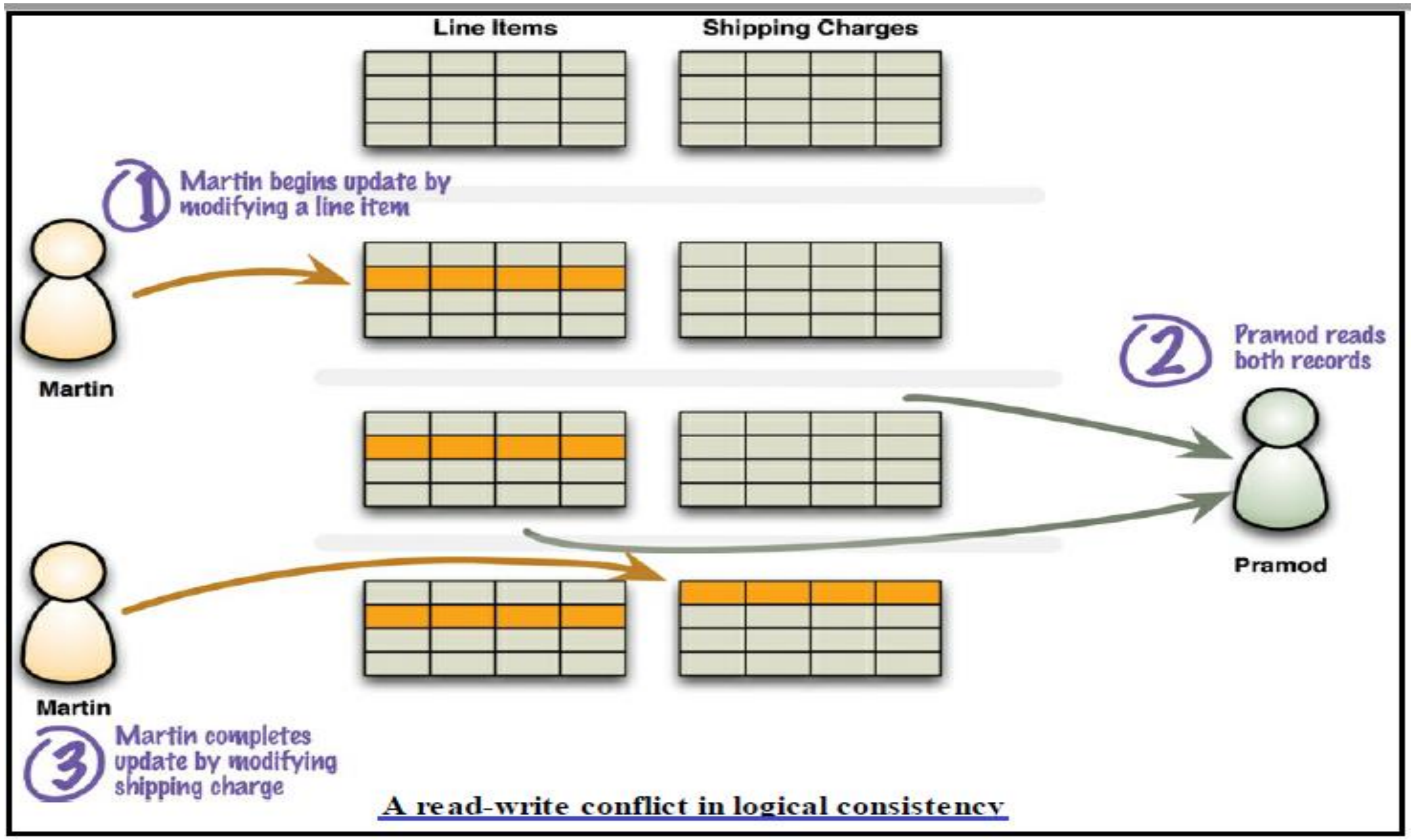
## Update Consistency

- People to prefer pessimistic concurrency because they are determined to avoid conflicts.
- 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
- 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

## Read Consistency

- Data store must 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**:

# Read Consistency


Line Items
Shipping Charges
1 Martin begins update by modifying a line item
Martin
2 Pramod reads both records
Pramod
Martin
3 Martin completes update by modifying shipping charge

A read-write conflict in logical consistency

- 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.

# Read Consistency

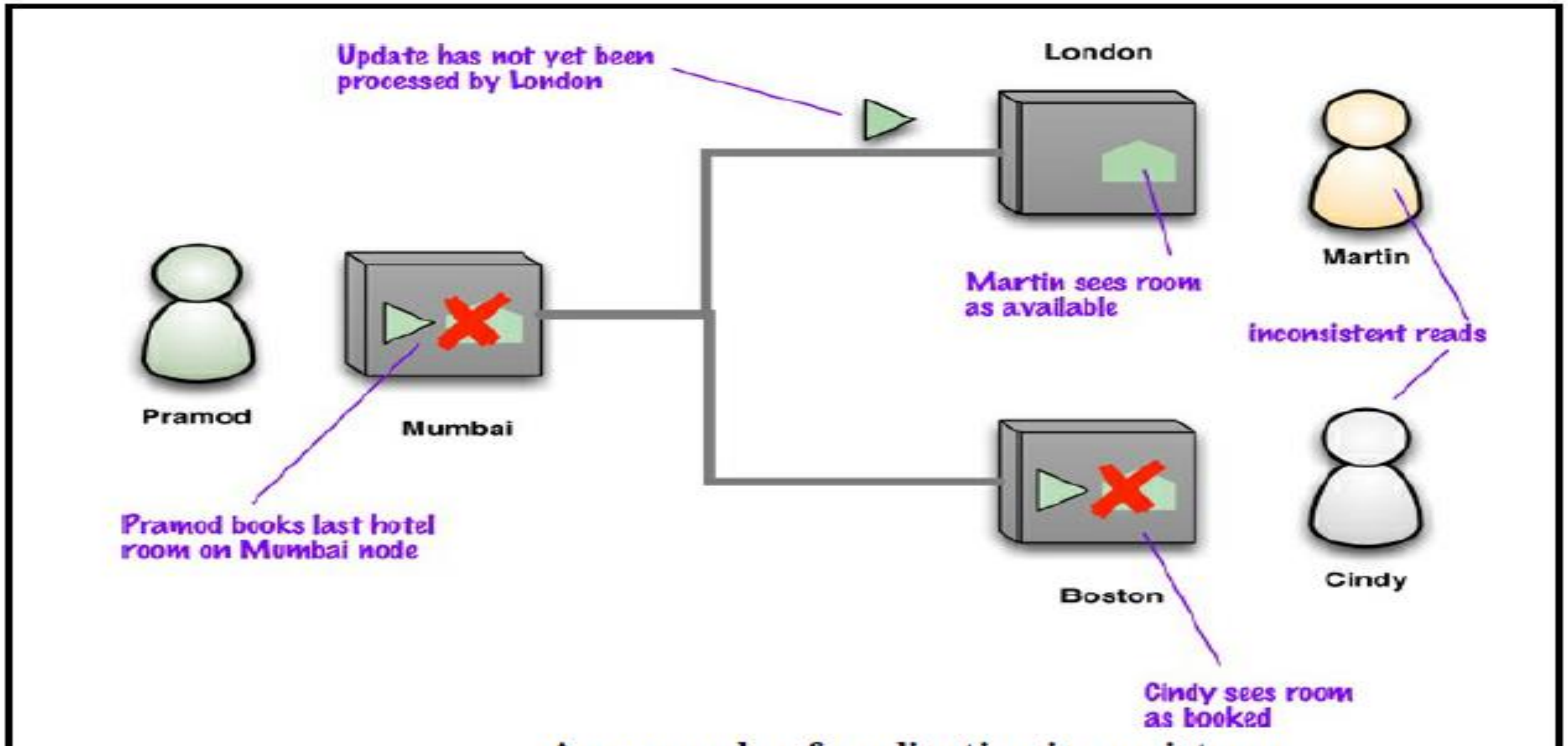
- A common claim we hear is that NoSQL databases don’t support transactions and thus can’t be consistent.
- Such claim is mostly wrong because it glosses over lots of important details.
- Our first clarification is that any statement about lack of transactions usually only applies to some NoSQL databases, 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.
- Have logical consistency within an aggregate but not between aggregates.
- Could avoid running into that inconsistency if the order, the delivery charge, and the line items are all part of a single order aggregate.

## Read Consistency

- ❑ Not all data can be put in the same aggregate
- ❑ 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.

# Read Consistency

- Another inconsistent read—but it's a breach of a different form of consistency we call **replication consistency**: ensuring that same data item has same value when read from different replicas
- the updates will propagate fully - situation is generally referred to as **eventually consistent**


Update has not yet been processed by London
London
Martin
Martin sees room as available
inconsistent reads
Pramod
Mumbai
Pramod books last hotel room on Mumbai node
Cindy
Boston
Cindy sees room as booked

An example of replication inconsistency

# Read Consistency

- The presence of an inconsistency window means that different people will see different things at the same time.
- 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.
- 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 is to provide **session consistency**: Within a user’s session there is read-your-writes consistency

## Read Consistency

- ❑ 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 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.

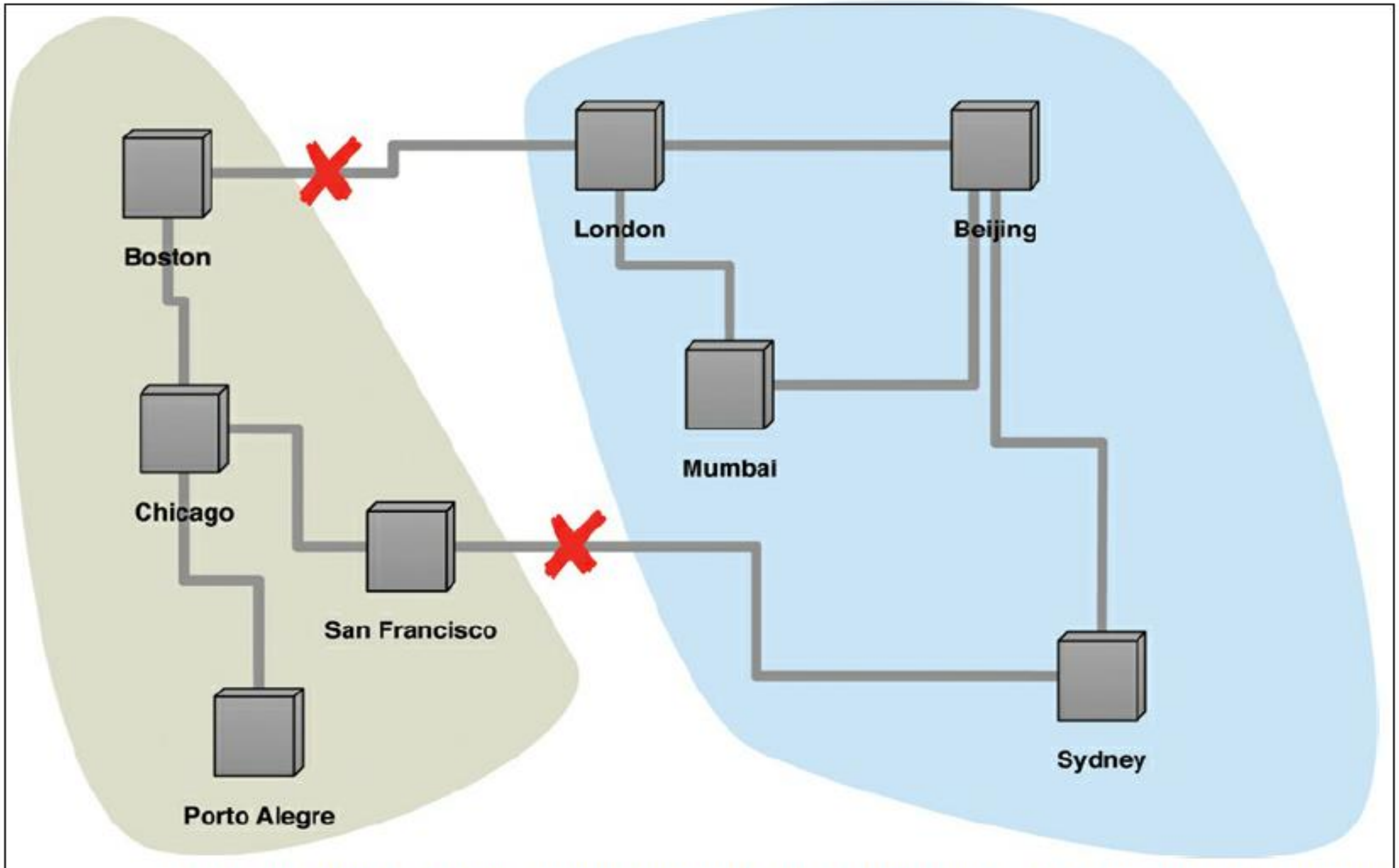
## Relaxing Consistency

- ❑ Consistency is a Good Thing—but, sadly, sometimes we have to sacrifice it.
- ❑ Often have to tradeoff consistency for something else.
- ❑ 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.
- ❑ Principal tool to enforce consistency is the transaction
- ❑ However, transaction systems usually come with the ability to relax isolation levels, allowing queries to read data that hasn't been committed yet
- ❑ In practice we see most applications relax consistency down from the highest isolation level (serialized) in order to get effective performance.

# The CAP Theorem

- ❑ In the NoSQL world it’s refer to the CAP theorem as the reason why you may need to relax consistency.
- ❑ It was originally proposed by Eric Brewer in 2000 and given a formal proof by Seth Gilbert and Nancy Lynch
- ❑ Basic statement of CAP theorem is that, given 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
- ❑ **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.
- ❑ **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

# The CAP Theorem



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

## The CAP Theorem

- ❑ 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.

# The CAP Theorem

- It is theoretically possible to have a CA cluster.
- 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"
- So a failed, unresponsive node doesn't infer a lack of CAP availability.

# The CAP Theorem

- ❑ 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.

# The CAP Theorem

- 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)
- 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.
- When Brewer introduced the notion of BASE, he saw the tradeoff between ACID and BASE as a spectrum, not a binary choice.

# Relaxing Durability

- 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?
- 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.

## Relaxing Durability

- Example of where 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.
- More important updates can force a flush to disk.

# Relaxing Durability

- 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.
- 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.

***Thank You !!!***