Data Management for Data Science

Corso di laurea magistrale in Data Science
Sapienza Università di Roma
2015/2016

NoSQL Databases: Aggregated DBs

Domenico Lembo, Riccardo Rosati

Dipartimento di Ingegneria Informatica Automatica e Gestionale A. Ruberti
Sapienza Università di Roma

Bibliographic References

- The main bibliographic reference for this part is:
  [SaFo13] *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Pramod J. Sadalage & Martin Fowler. Addison Wesley. 2013

NoSQL databases: Aggregated DBs

- NoSQL data models
- Key-value, document, column databases
- Distribution models
- Consistency
- Map-Reduce

NoSQL: beyond graph databases

- So far we have investigated Graph Databases, mainly for their ability of providing
  - schemaless modeling of data
  - native treatment of relationships between pieces of information
- The above characteristics make them particularly suited at handling data with complex relationships, in particular in those contexts in which the domain dynamics make solutions based on the classical relational model not effectively and efficiently applicable (e.g., user connections' in a social networks, recommendation systems, geospatial applications, ecc.).
- We also have investigated an interesting use of graph databases specified through the RDF W3C standard for data (and knowledge) sharing at the web scale.

NoSQL: completing the picture

- Graph databases are only a particular family of databases that we can classify as belonging to the "NoSQL movement"
- Also, graph databases generally present only some of the characteristics that are typical of NoSQL solutions, and which we summarize below (even though there is no generally accepted definition of NoSQL in the literature):
  - schemaless
  - not using SQL
  - generally open-source (even though the NoSQL notion is also applied to closed-source systems)
  - generally driven by the need to run on clusters (but graph databases do not typically fall in this class)
  - generally not handling consistency through ACID transactions (but notice that graph databases instead do it)

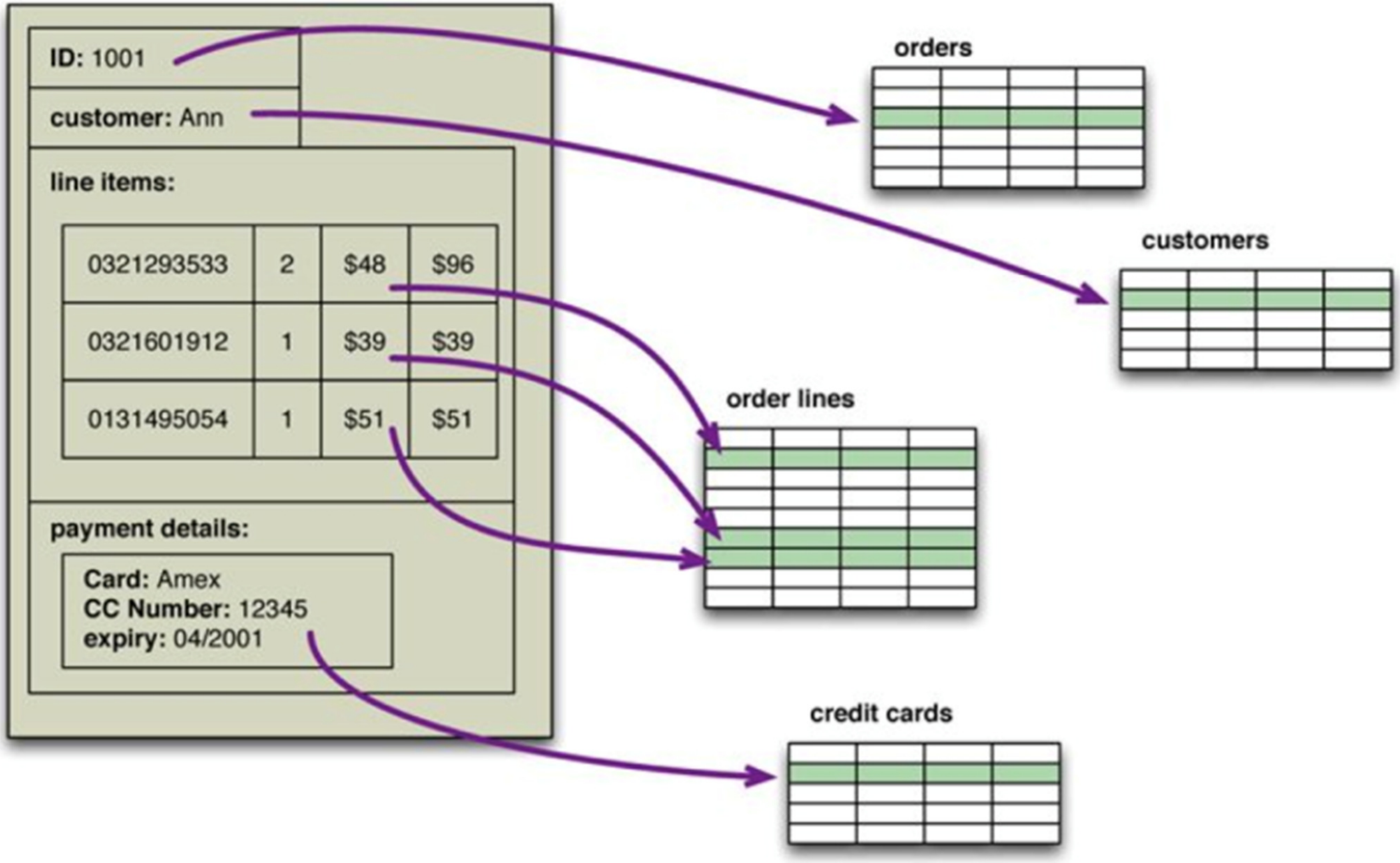
NoSQL: Aggregate data models

- Besides Graph databases other three categories are widely used in the NoSQL ecosystem to classify data models adopted by NoSQL solutions:
  - **key-value**
  - **document**
  - **column-family**
- They share a common characteristic of their data models which we will call **aggregate orientation**[SaFo13]
  - Aggregate is a term that comes from Domain-Driven Design (DDD) (http://dddcommunity.org/). In DDD, an aggregate is a collection of related objects that we wish to treat as a unit. In particular, it is a unit for data manipulation and management of consistency

Aggregate data models

- **The relational model divides the information** that we want to store **into tuples** (rows): this is a very simple structure for data (which somehow is the key of the success of the relational model and the cause of the relational dominance we experienced from the late 70s to the first years of 2000s)
- **Aggregate orientation** takes a different approach. It **recognizes** that often **you want to operate on data in units that have a more complex structure**.
- 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 well 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; according to [SaFo13] we use here the term *aggregate.*

Aggregate data models



An order, which looks like a single aggregate

Aggregate data models

- A natural question is now which are the **main motivations** at the basis of the rise of aggregate data models and tools supporting them
- Some of them coincide with the motivations that originated the development of the Big Data ecosystem, and which we already discussed (recall the three v's, etc.)
- According to [SaFo13], two main aspects however should be emphasized:
  - Dealing with 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
  - Also, it may help **solving the impedance mismatch problem**, i.e., the difference between the relational model and the in-memory data structures (see previous figure)

Attack of the clusters

- The 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
- Coping with the increase in data and traffic required more computing resources
- **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

Attack of the clusters

- **The alternative is to scale out, i.e., 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!

Attack of the clusters

- Relational databases **could also be run as separate servers** for different sets of data, effectively ***sharding*** the database, (i.e., data are physically segmented on various storage nodes)
- 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 for each bit of data
- Also, we lose any querying, referential integrity, transactions , or consistency controls that cross shards
- Deciding the **granularity of sharding is a very difficult issue!**
- **Aggregate orientation fits well with scaling out** because **the aggregate is a natural unit to use for distribution**
- On the other hand, **NoSQL aggregate databases generally adopt a relaxed notion of consistency,** with respect to the classical one used in the relation world which is based on ACID transactions. Also, **slicing aggregates** for more fine grained access to them **may become very difficult**

Impedance mismatch

- 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

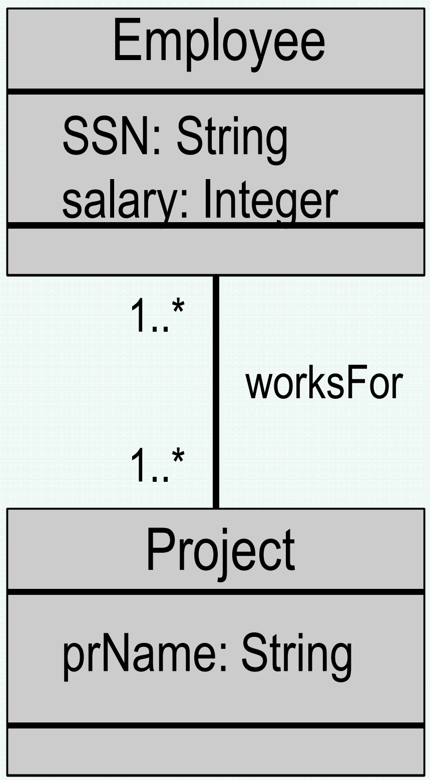
- That decade was marked with the growth of object-oriented programming languages, and with them came object-oriented databases

- However, while object-oriented languages succeeded in becoming the major force in programming, object-oriented databases were not successful: Relational databases remained the major technology for data storage, being them highly consolidated, well-known, optimized, and, above all, based on standard language (SQL)

- Thus, impedance mismatch remained an issue: Object-relational mapping frameworks like Hibernate or iBatis have been proposed that make it easier, but are not suited for those (frequent) scenarios in which many applications rely on the same (integrated) database. Also, query performance in general suffers under these frameworks .

Impedance mismatch - example

in-memory data structure



Actual data is stored in a DB:

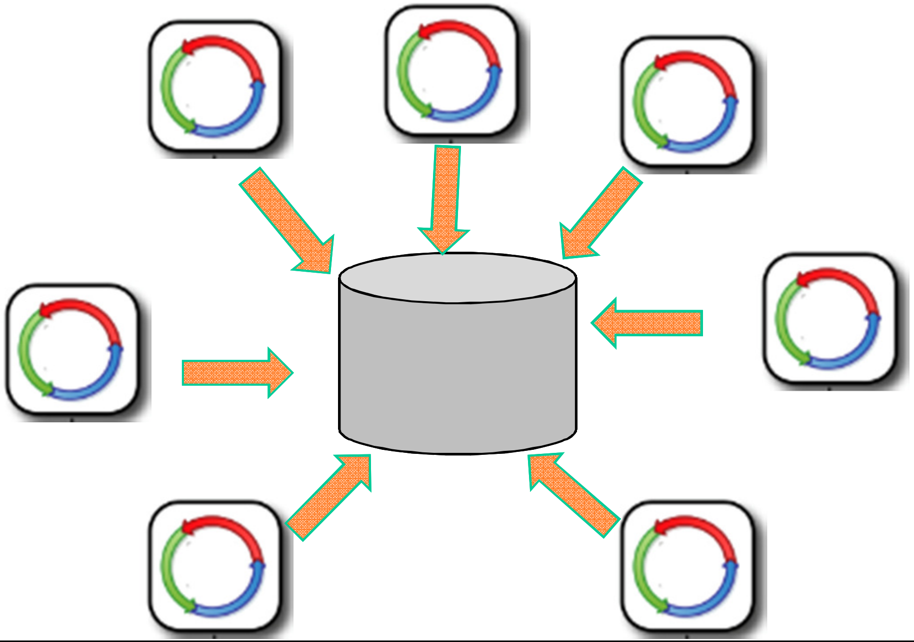
D2[*Code*: Int, *Salary*: Int, *SSN*: String]
Employee's Code with salary and SSN

D1[*Code*: Int, *PrName*: String]
Employees and Projects they work for

Conceptually:
– An Employee is identified by her *SSN*.
– A Project is identified by its *name*.
Thus,
(i) an employee should be created from her *SSN*;
(ii) a project should be created from its *PrName*

SQL as an integration mechanism

- The primary factor that made relational databases more successful over OO databases is probably the role played by **SQL as an integration mechanism between applications**
- In this scenario, multiple applications store their data in a common, integrated database. This improves communication because all the applications are operating on a consistent set of persistent data



Integration vs application databases

- There are downsides to shared database integration
  - ✓ A structure that is designed to integrate many applications ends up being more complex
  - ✓ Changes to data by different applications need to be coordinated
  - ✓ different applications have different performance needs, thus call for different index structures
  - ✓ complex access control policies
- A different approach is to treat your database as an **application database**

Application databases

- An application database is only directly accessed by a single application, which makes it much easier to maintain and evolve
- Interoperability concerns can now shift to the interfaces of the application:
  - ✓ During the 2000s we saw a distinct shift to web services, where applications would **communicate over HTTP** (cf. work on Service-oriented Architecture).
- If you communicate with SQL, the data must be structured as relations. However, **with a service, you are able to use richer data structures**, possibly with nested records and lists.
- These are usually represented as documents in XML or, more recently, JSON (JavaScript Object Notation), a lightweight data-interchange format.

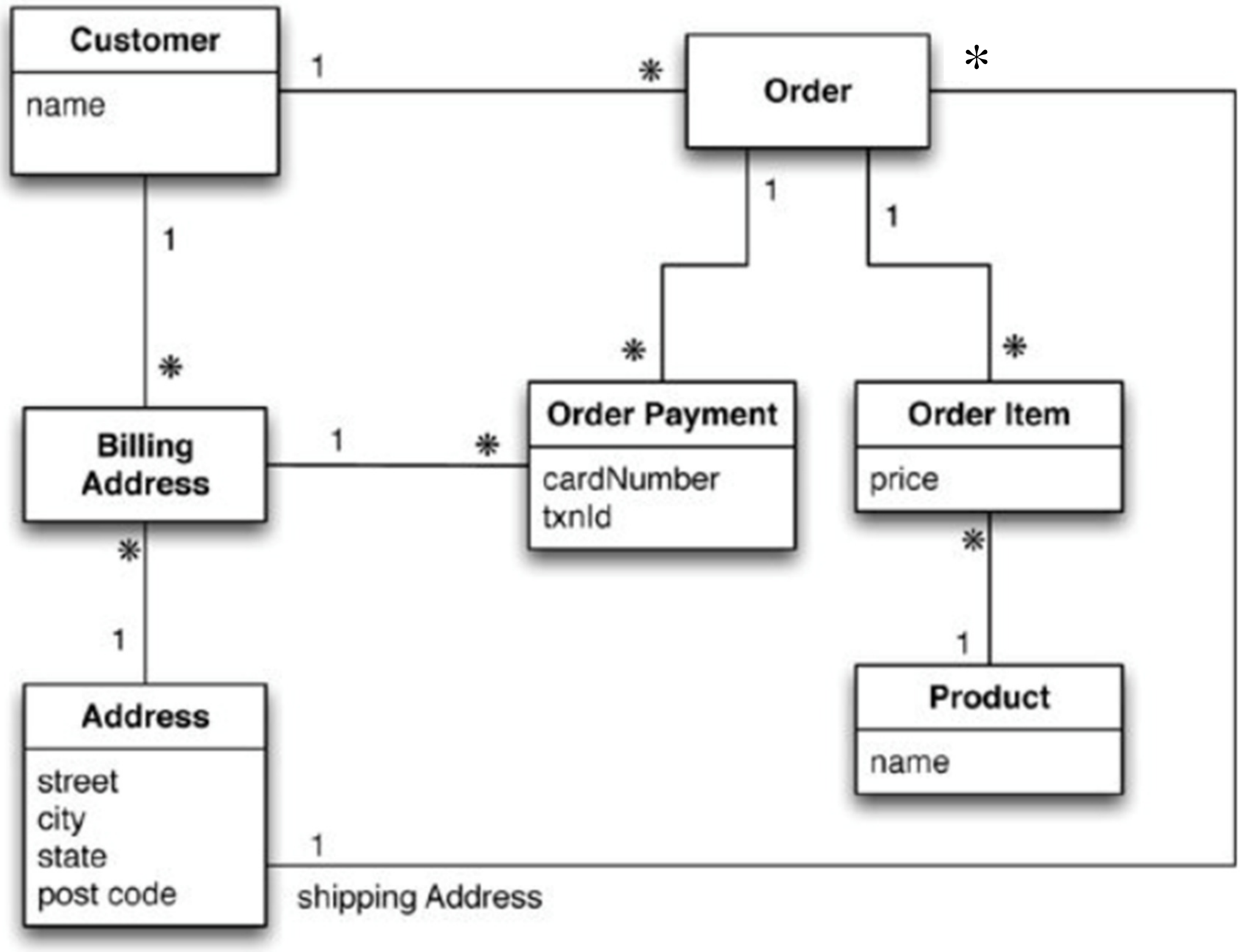
Application databases

- Since using application databases 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

Note: *On the other hand, when each application has its own database, the risk arises that data may become non-aligned, mutually inconsistent, and difficult to access in an integrated fashion. All these aspects should be indeed managed at the application level, which does not always take care of them.*

Example of Relations and Aggregates



Relational database perspective: no aggregates

Example of Relations and Aggregates

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

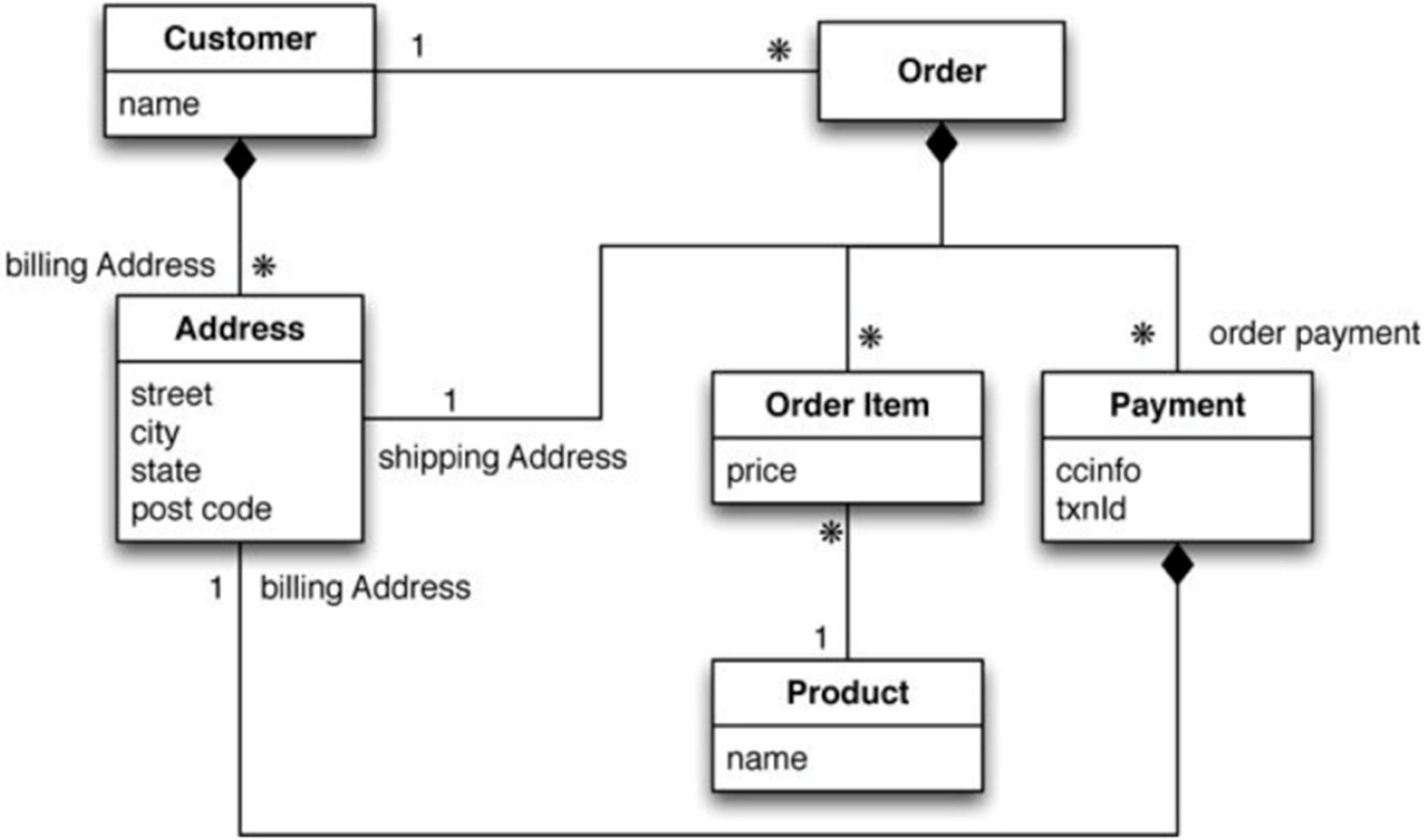
OrderItem			
Id	OrderId	ProductId	Price
100	99	27	32.45

Address	
Id	City
77	Chicago

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

Note: for simplicity, only interesting attributes for the instance at hand of the Address *relation are represented. By default, each entity is assigned with a code*

Example of Relations and Aggregates



Note: Address is strongly aggregated into Customer (implicit cardinality 0..1). Order is a strong aggregation of Address, OrderItem and Payment. Payment is a strong aggregation of Addresses (strong aggregation means that a single istance of Address is aggregated into a single instance of Customer, or Order, or Payment, but not in more than one)

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"}
  }
  ],
}
```

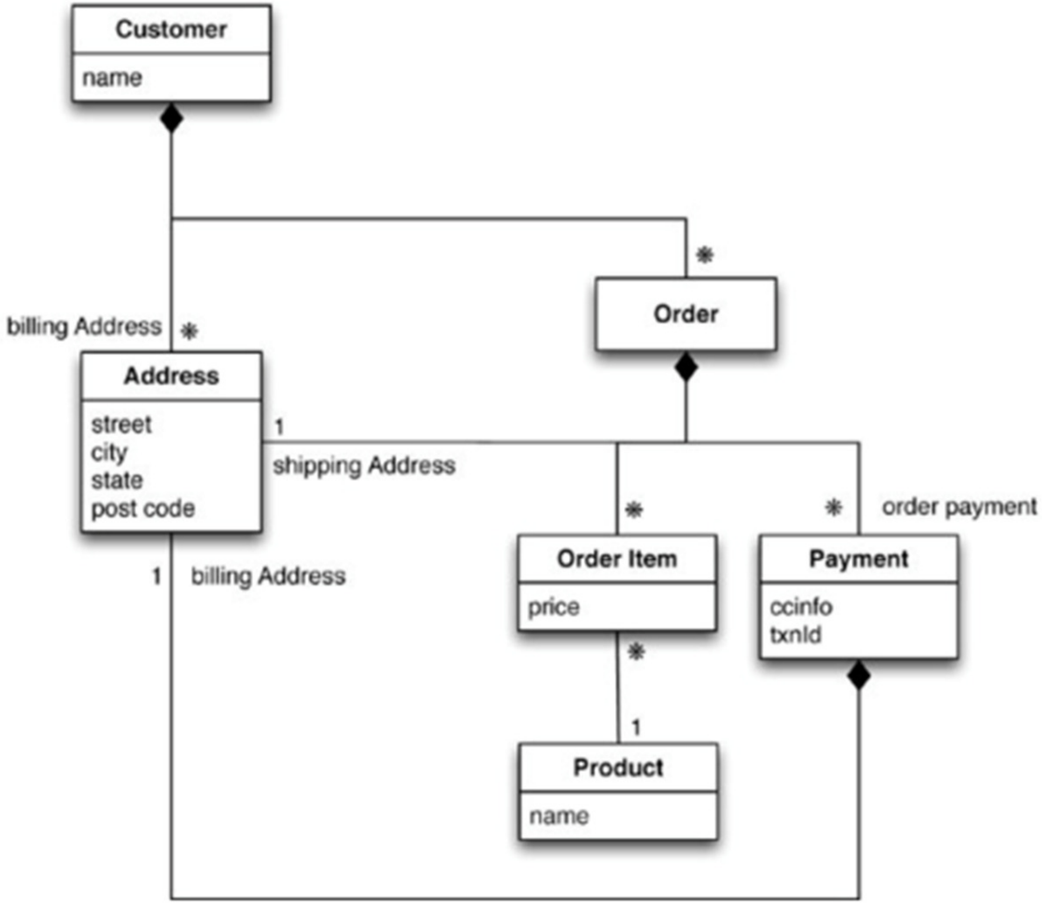
There are two main aggregates: *customer* and *order*

The customer contains a list of billing addresses and a name; the order contains a list of order items, a shipping address, and a list of payments. Each payment 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 is treated as a value and copied each time.

The link between the customer and the order isn't an aggregation. However, we've shown the product name as part of the order to minimize the number of aggregates we access during a data interaction

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"}]
```

....

An alternative way of aggregating data

Consequences of Aggregate Orientation

- The fact that an order consists of order items, a shipping address, and a payment 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 the knowledge about an aggregate structure to help it store and distribute the data
- Aggregation is however, not a logical data property: It is all about how the data is being used by applications -- a concern that is often outside the boundary of data modeling
- Also, an aggregate structure may help with some data interactions but be an obstacle for others (in our example, to get to product sales history, you'll have to dig into every aggregate in the database)
- The clinching reason for **aggregate orientation** is that it **helps greatly with running on a cluster**!

Consequences of Aggregate Orientation

- Aggregates have an important consequence for transactions.
- Relational databases allow you to manipulate any combination of rows from any tables in a single (ACID) transaction (i.e., Atomic, Consistent, Isolated, and Durable)
- It's often said that NoSQL databases don't support ACID transactions and thus sacrifice consistency. This is however not true for graph databases (which are, as relational database, aggregate-agnostic)
- In general, its 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 **is part of the consideration for deciding how to divide up our data into aggregates**

NoSQL databases: Aggregated DBs

- NoSQL data models
- Key-value, document, column databases
- Distribution models
- Consistency
- Map-Reduce

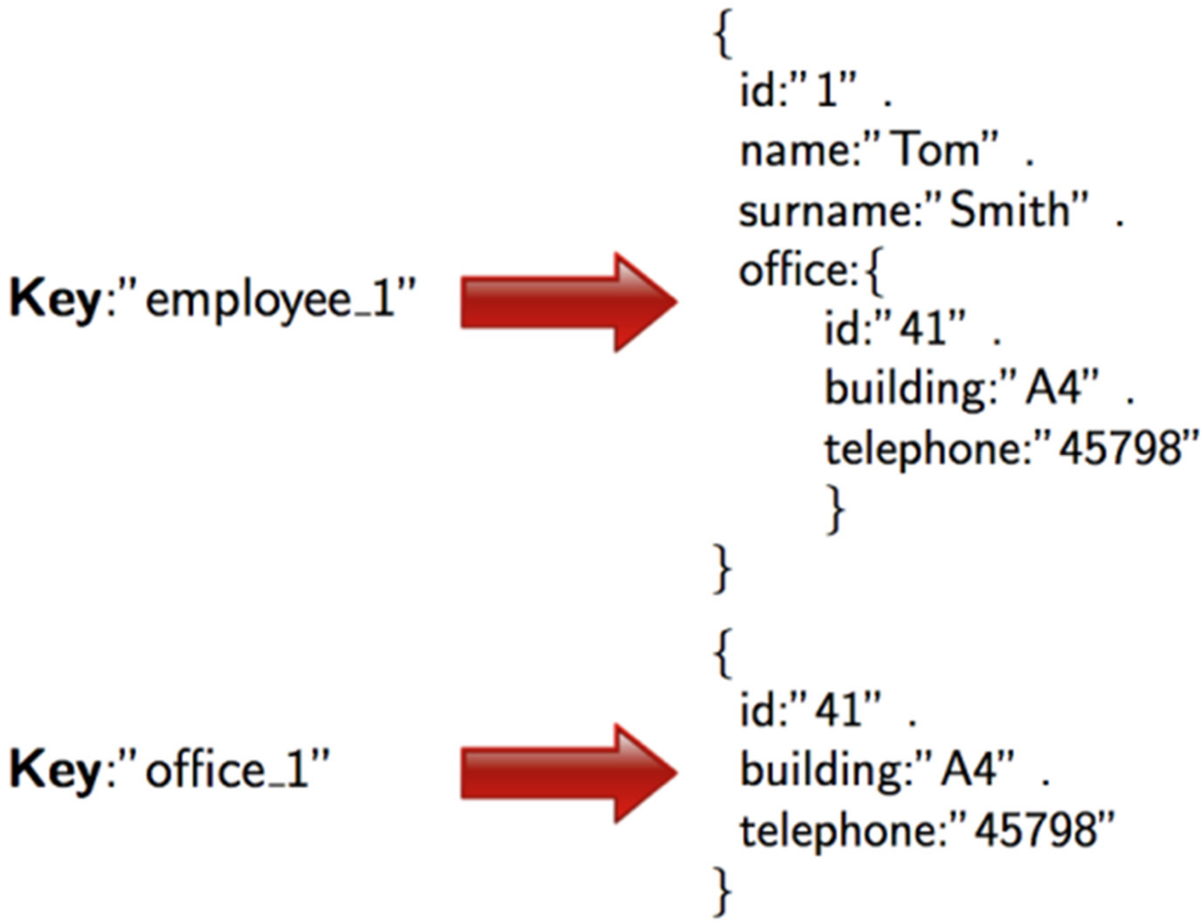
Key-Value and Document Data Models

- We said earlier on that key-value and document databases were strongly aggregate-oriented
- **In a key-value database, the aggregate is opaque** to the database: just some big blob of bits. The advantage of opacity is that **we can store whatever we like in the aggregate**. It is the responsibility of the application to understand what was stored. Since key-value stores always use primary-key access, they generally have great performances, Descended as they are from Amazon's Dynamo database—a platform designed for a nonstop shopping cart service—Key-values stores essentially act like large, distributed hashmap data structures.
- In contrast, **a document database is able to see a structure in the aggregate**, but **imposes limits on what we can place in it**, defining allowable structures and types. In return, however, **we get more flexibility when accessing data**.

Key-Values: example

Key	Value
employee_1	name@Tom-surn@Smith-off@41-buil@A4-tel@45798
employee_2	name@John-surn@Doe-off@42-buil@B7-tel@12349
employee_3	name@Tom-surn@Smith
office_41	buil@A4-tel@45798
office_42	buil@B7-tel@12349

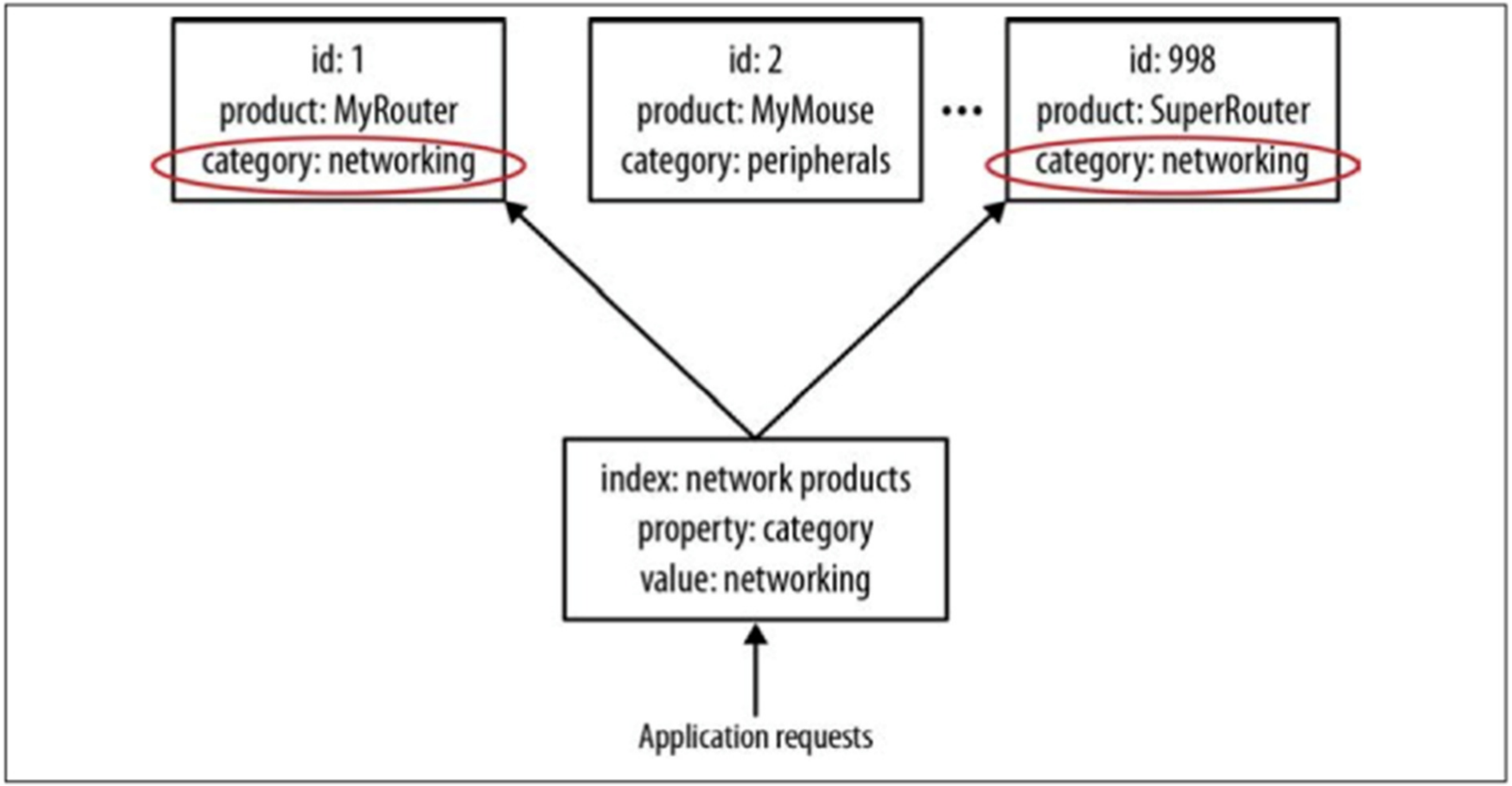
Documents: example (JSON format)



Key-Value and Document Data Models

- With a key-value store, we can only access an aggregate by lookup based on its key
- At the simplest level, even in document databases documents can be stored and retrieved by ID (as key-values stores). however, in general, 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 general, indexes are used to retrieve sets of related documents from the store for an application to use.
- As usual, indexes speed up read accesses but slow down write accesses, thus they should be designed carefully.

Indexes on Document Data Models



For example, in an ecommerce scenario, we might use indexes to represent distinct product categories so that they can be offered up to potential sellers.

Key-Value and Document Data Models

- 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 look-up. Databases classified as key-value databases may allow you structures for data beyond just an opaque. For example, Redis allows you to break down the aggregate into lists or sets, Riak allows you to put aggregates into buckets.
- Some of the popular key-value databases are Riak, Redis (often referred to as Data Structure server), Memcached DB, Berkeley DB, HamsterDB (especially suited for embedded use), Amazon DynamoDB (not open-source), Project Voldemort (an open-source implementation of Amazon DynamoDB)
- Some of the popular document databases are MongoDB, CouchDB, Terrastore, OrientDB (which is also a graph DBMS), RavenDB, but also Lotus Notes, which adopts document storage.

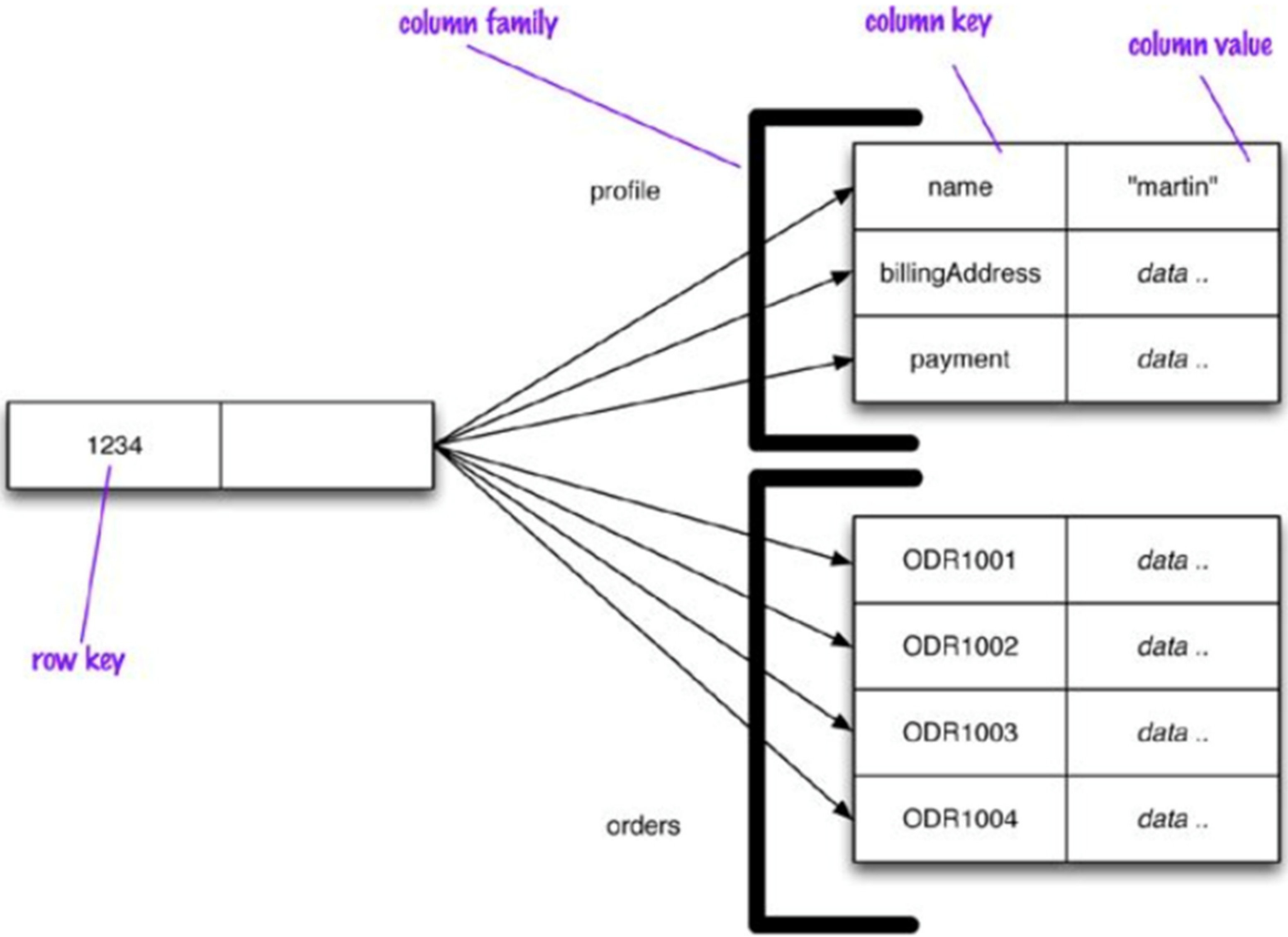
Column Family Stores

- Column family stores are modeled on Google's BigTable. The data model is based on a sparsely populated table whose rows can contain arbitrary columns.
- Some popular Column Family stores are Cassandra, Hbase, Hypertable, and Amazon SimpleDB
- *Note: These databases with a bigtable-style data model are often referred to as column stores, and not column family stores, but that name has been around for a while to describe a different object: Pre-NoSQL column stores, such as C-Store or MonetDB, were happy with SQL and the relational model. The thing that made them different was the way in which they physically stored data, based on columns rather than on rows as a unit for storage (this storage system is particularly suited to speed up read accesses)*

Column Family Stores

- The column-family model can be seen 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
  - This row aggregate is itself formed of a map of more detailed values. These second-level values are referred to as columns, each being a key-value pair
- Columns are organized into column families. Each column has to be part of a single column family (data for a particular column family will be usually accessed together)
- Each row identifier (i.e., first-level key) is unique in the context of a single Column Family.

Column Family Stores



Column Family Stores

Two ways to think about how the data is structured:

- **Row-oriented**: Each row is an aggregate (for example, customer with the ID of 1234) with column families representing useful chunks of data (profile, order history) within that aggregate

- **Column-oriented:** Each column family defines a record type (e.g., customer profiles) with rows for each of the records. You then think of a row as the join of records in all column families. Column Families can be then to some extent considered as tables in RDBMSs. Unlike table in RDBMSs, a Column Family can have different columns for each row it contains

Column Family Stores: Cassandra

- The terminology used so far is as established by Google Bigtable and HBase, but *Cassandra* looks at things slightly differently:
  - What we have called column family according to the Bigtable terminology is somehow equivalent to Cassandra supercolumns, i.e., columns that contain nested columns.
  - We can still use the term column family in Cassandra, and think of column families as tables, but now, each row in the table, besides possibly having different columns with respect to other rows, presents columns aggregated in supercolums. Also, a row in Cassandra only occurs in one column family.

Column Family Stores: Cassandra terminology

ColumnFamily: Employees

Key	id	name	surname	office		
				id	buil.	tel.
employee_1	1	Tom	Smith	41	A4	45798

Key	id	name	surname
employee_3	3	Anna	Smith

Key	id	name	surname	office	
				id	buil.
employee_2	2	John	Doe	42	B7

Aggregate DBs: Wrapping up

- All aggregate data models are based on the notion of an aggregate indexed by a key that you can use for lookup. Within a cluser, all the data for an aggregate should be stored together on one node. The aggregate also acts as the atomic unit for management
- The key-value data model treats the aggregate as an opaque whole (no access to portion of an aggregate is allowed). Great performances are allowed but the aggregate has to be understood at the application level
- The document model makes the aggregate transparent, 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
- 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

Schemaless databases

- NoSQL databases are schemaless:
  - 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.
- This has various advantages:
  - Without a schema binding you, you can easily store whatever you need, and change your data storage as you learn more about your project
  - You can easily add new things as you discover them
  - A schemaless store also makes it easier to deal with **nonuniform** data: data where each record has a different set of fields (limiting sparse data storage)

Schemaless databases

- Schemalessness is appealing, but it brings some problems of its own
- Indeed, whenever we write a program that accesses data, that program almost always relies on some form of **implicit schema**: 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
- Having the implicit schema in the application means that in order to understand what data is present you have to dig into the application code. Furthermore, the database remains ignorant of the schema: it cannot use the schema to support the decision on how to store and retrieve data efficiently. Also, it cannot impose integrity constraints to maintain information coherent

Schemaless databases

- Since the implicit schema is into the application code that accesses it, the situation becomes problematic if multiple applications 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 (e.g., different sections of a document, different column families, etc.)
- **Remark:** if you need to change your aggregate boundaries in aggregate databases, the data migration is as complex as it is in the relational case (remember also that, even though not frequent, relational schemas can be changed at any time with standard SQL commands).

(Materialized) Views

- Although NoSQL databases don’t have views as relational databases, they may have precomputed and cached queries, and they use the term “materialized views” to describe them
- This is particularly useful for those applications that have to deal with some queries that don’t fit well with the aggregate structure
- There are two basic strategies to manage materialized views
  - update the materialized view at the same time you update the base data for it (this is useful if you have more read than write accesses)
  - run batch jobs to update the materialized views at regular intervals (of course, in this case some temporal windows exist in which data in the materialized views may be not aligned).

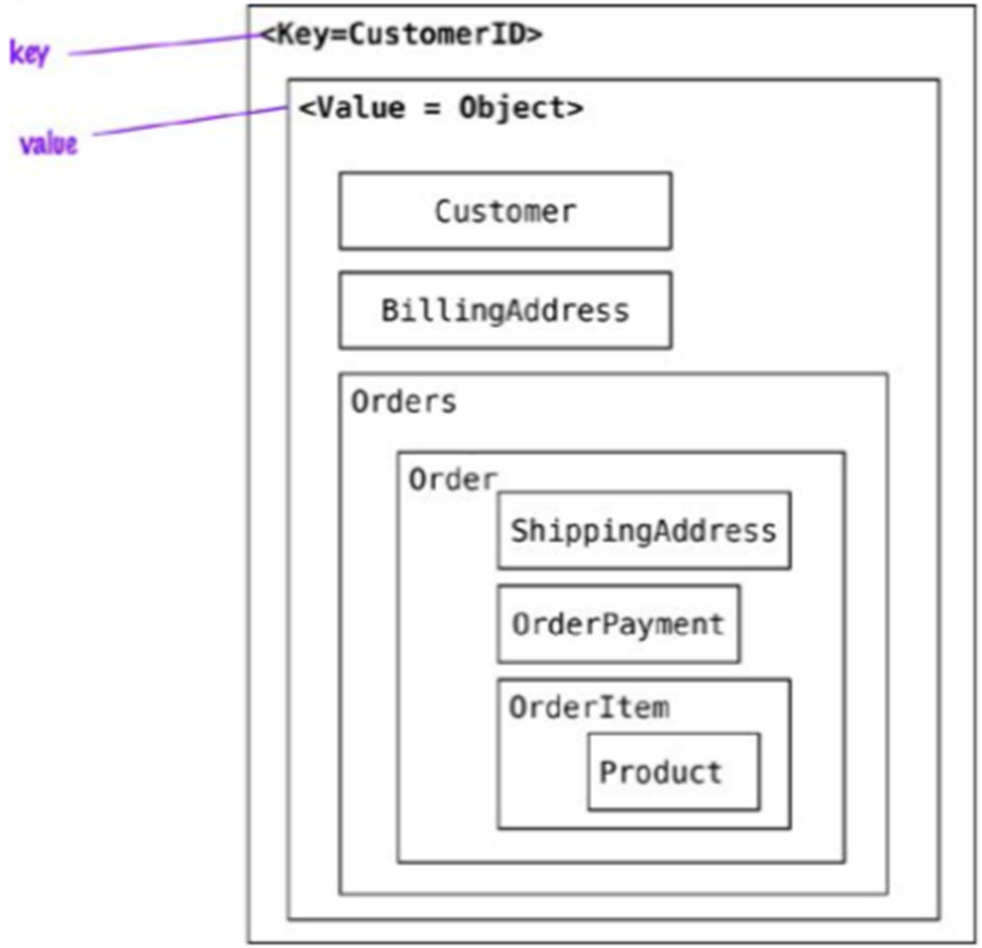
in both approaches, strategies for incremental updates of view are often used.

Data Modeling

- Despite several NoSQL tools have been developed in the last years, and various technical solutions have been proposed so far, to date no methodologies have been developed to guide the database designer in the modeling of a NoSQL database
- This contrasts with the well established methodologies available for the design of a relational database
- This is however justified by the fact that NoSQL data models and technologies are still in their infancy
- Methodologies need thus to be devised to both (i) model data (e.g., decide the form of aggregates in aggregate DBs), and (ii) distribute data on a cluster (in this respect, we notice that no consolidated methodologies exist even for distributing relational databases over several storage nodes in a network)
- In what follows we limit to present some general considerations on data modeling with the help of an example

Data Modeling

- When modeling data aggregates we need to consider how the data is going to be read (and what are the side effects with the chosen aggregates)
- **Example:** Let' s start with the model where all the data for the customer is embedded using a key-value store



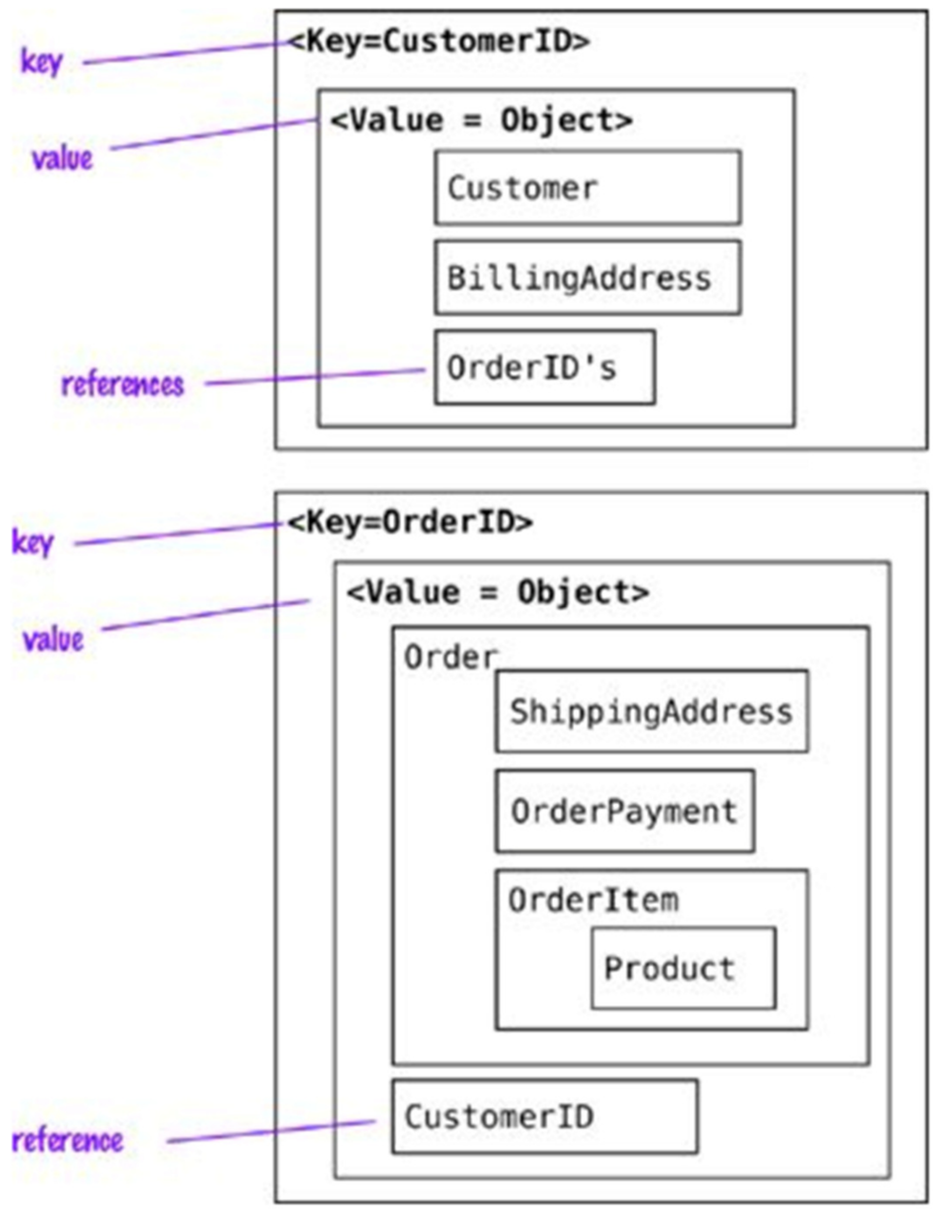
- The application can read the customers information and all the related data by using the key
- To read the orders or the products sold in each order, the whole object has to be read and then parsed on the client side

Data Modeling

When references are needed we should change the data for the key-value store to split the value object into Customer and Order objects and then maintain these objects reference

We can now find the orders independently from the Customer, and access then the customer using the CustomerID reference in the Order, whereas with the OrderId 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 for every new Order



Data Modeling

In document stores, since we can query inside documents, we can find all Orders for the Customer even removing references to Orders from the Customer object. This change allows us to not update the Customer object when 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"}
}
```

Data Modeling

- 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 Businees Intelligence** or **Real Time Analytics**: enterprises do not have to rely on end-of-the-day batch to populate data warehouses tables and generate analytics.
- Of course, only pre-packed analyses are possible through this approach

document store modeling:

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

Data Modeling

Remark:

- Since document data stores allow you to query by attributes inside the document, searches such as "*find all orders that include the 'Divina Commedia' product*" are possible

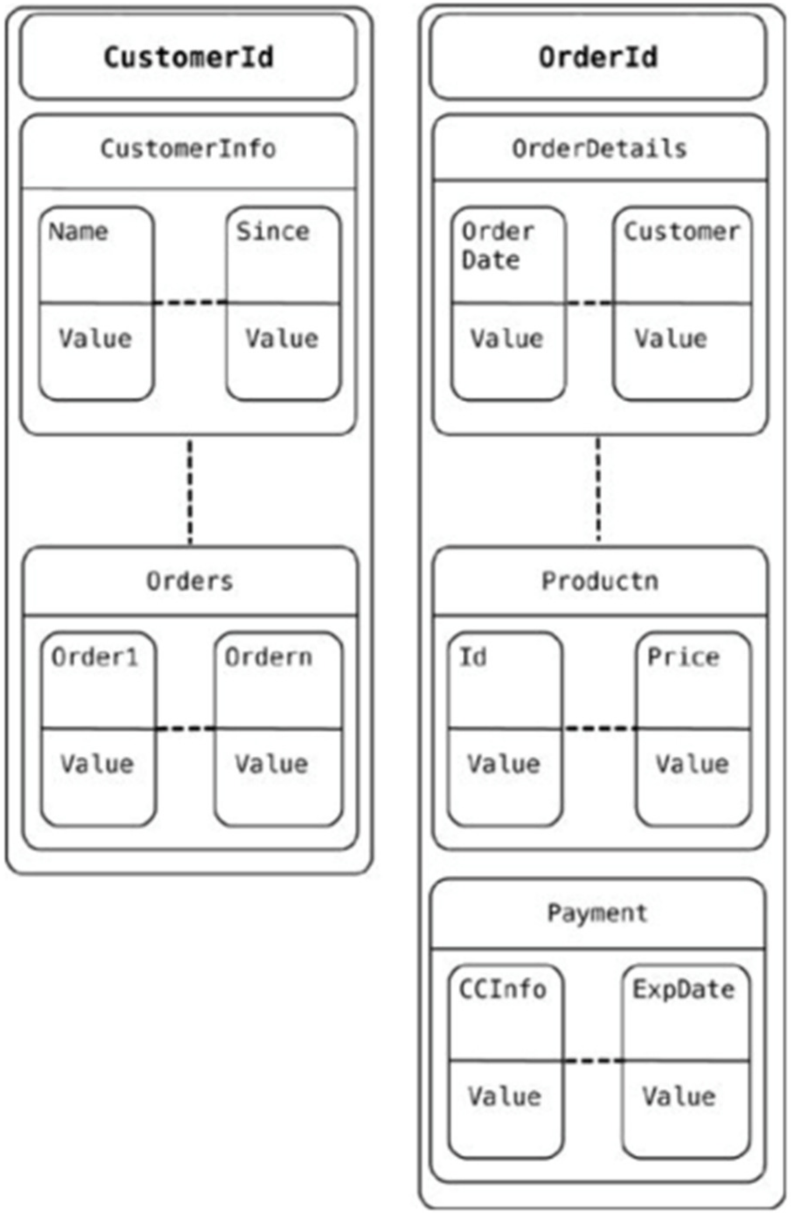
- Creating an aggregate in a document store of product and orders it belongs to is therefore not necessary for obtaining the result we are looking for, but rather for optimizing the way we obtain it.

Data Modeling

- When using the column families to model the data, we can model the schema a little more. Obviously, there are multiple ways to model the data

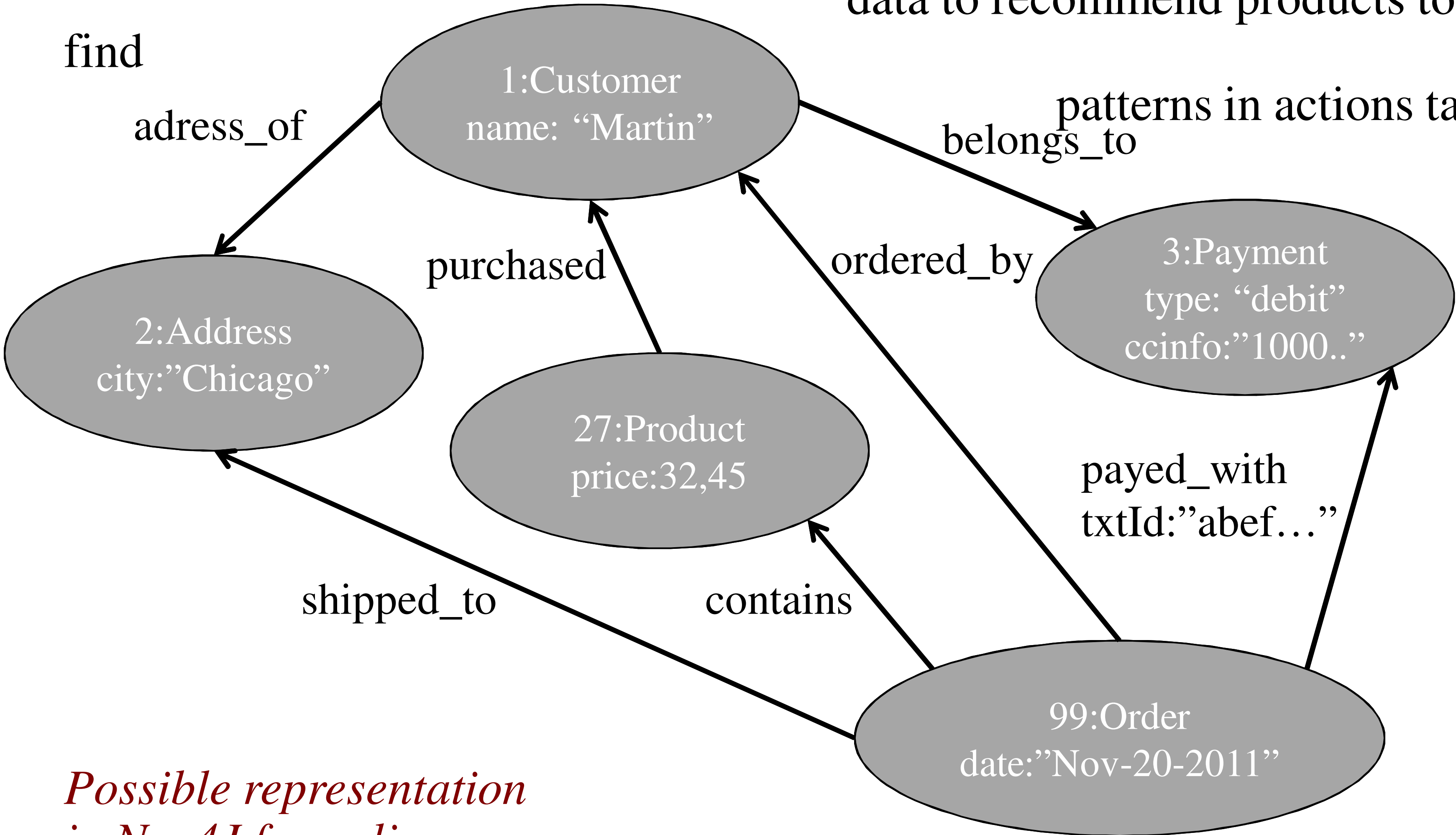
- In our example, one way is to store the Customer and Order in different column families

- 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



Data Modeling

When using graph databases to model the same data, we model all objects as (typed) nodes and relations within them as (typed) edges; both nodes and edges may have properties (key/value pairs). This is especially convenient when you need to use the data to recommend products to users or to find patterns in actions taken by users.



Possible representation in Neo4J formalism

NoSQL databases: Aggregated DBs

- NoSQL data models
- Key-value, document, column databases
- Distribution models
- Consistency
- Map-Reduce

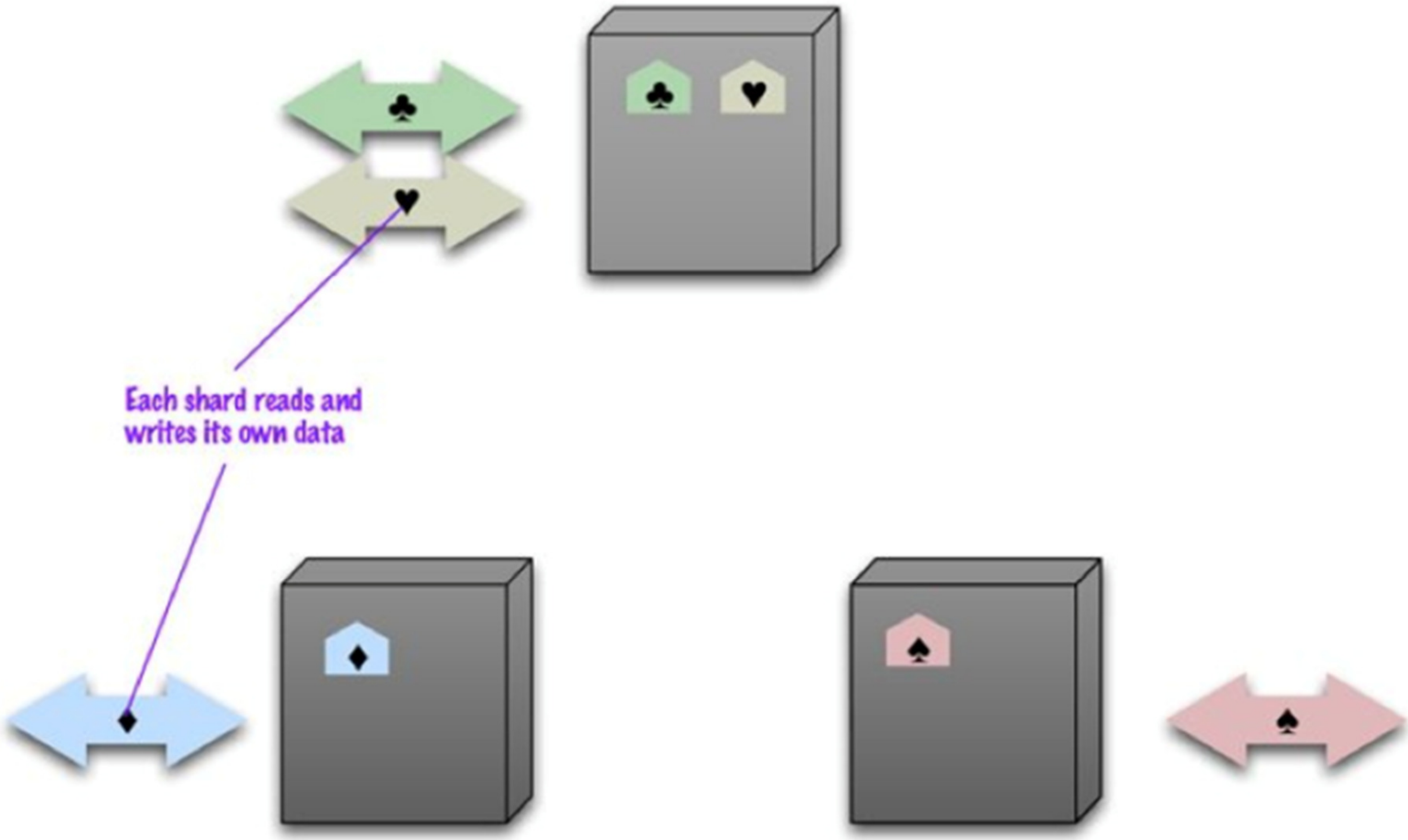
Distribution Models

- As said, the primary driver of interest in NoSQL has been its ability to run databases on a large cluster
- In particular, aggregate orientation fits well with scaling out because the aggregate is a natural unit to use for distribution
- Let us now have a look to various models for data distribution
  - **Sharding**
  - **Master-slave replication**
  - **Peer-to-peer replication**

Note: What follows is a brief overview. More on this topic is addressed in the course “Distributed Systems” (Prof. R. Baldoni)

Sharding

- Often, a busy data store is busy because different people are accessing different parts of the dataset. In these circumstances we can support horizontal scalability by putting different parts of the data onto different servers. This technique is called *Sharding*



Sharding

- Two main issues arise in Sharding:
  1. how to clump the data, so that one user mostly gets her data from a single server
  2. how to arrange single data clumps on the nodes to provide the best data access
- As for point 1, we recall that we generally design aggregates to combine data that are commonly accessed together. So aggregates leap out as an obvious unit of distribution
- As for point 2, there are several factors that can help improve performance. If you know that most accesses of certain aggregates are based on a physical location, you can place the data close to where it is being accessed. Another factor is trying to arrange aggregates so they are evenly distributed across the nodes which all get equal amounts of the load. This may vary over time

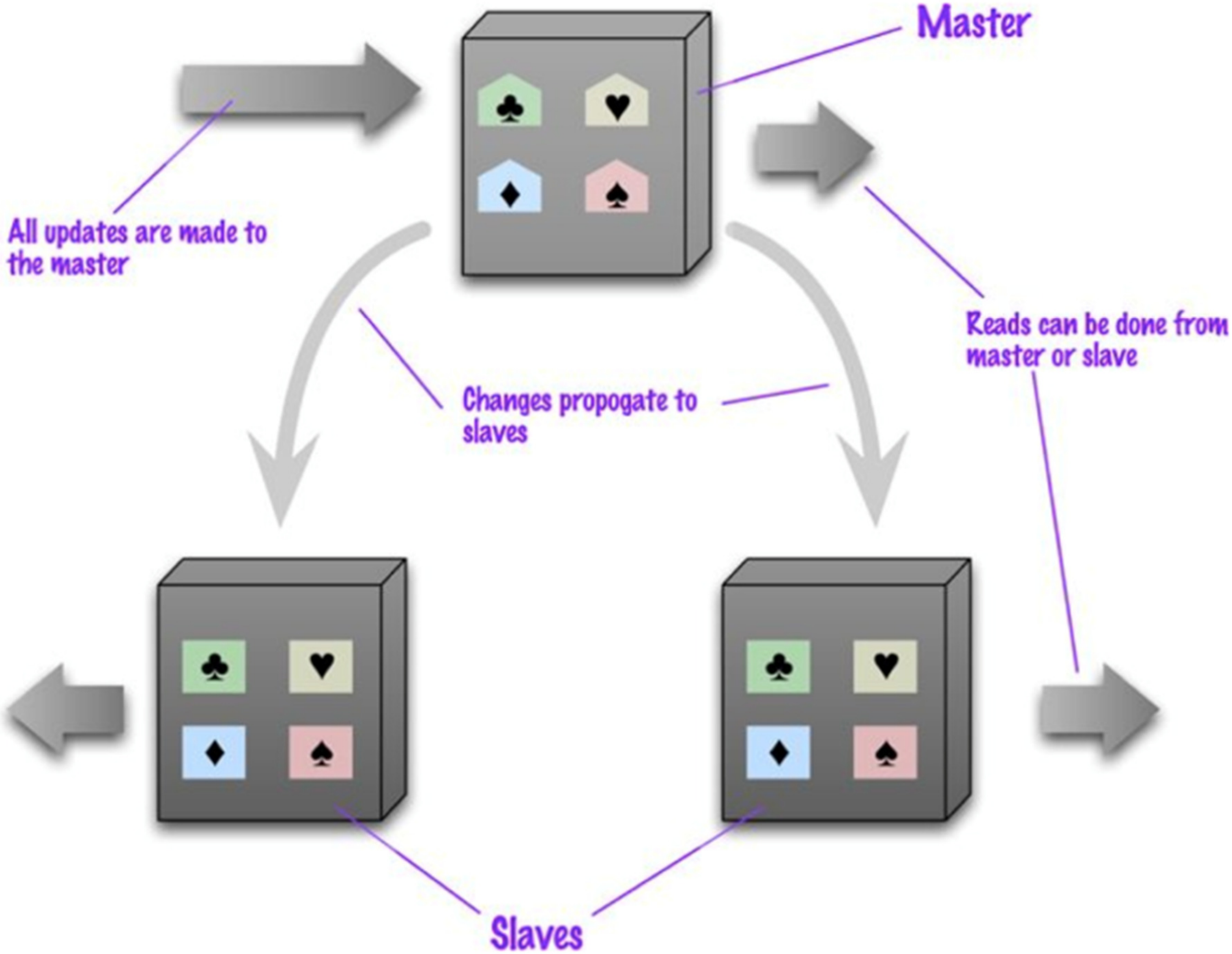
Sharding

- Sharding is particularly valuable for performance because it can improve both read and write performance
- Sharding however does little to improve resilience when used alone: Although the data is on different nodes, a node failure makes that shards data unavailable just as surely as it does for a single-server solution. The resilience benefit it does provide is that *only* the users of the data on that shard will suffer, which is however not that much! Furthermore, we notice that clusters often use less reliable machines than those adopted in single-server solutions, and therefore node failures can be more frequent. So in practice, sharding alone is likely to decrease resilience
- 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

Master-slave replication

- With master-slave distribution, you replicate data across multiple nodes
- One node is designated as the master, or primary, and is the authoritative source for the data, 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



Master-slave replication

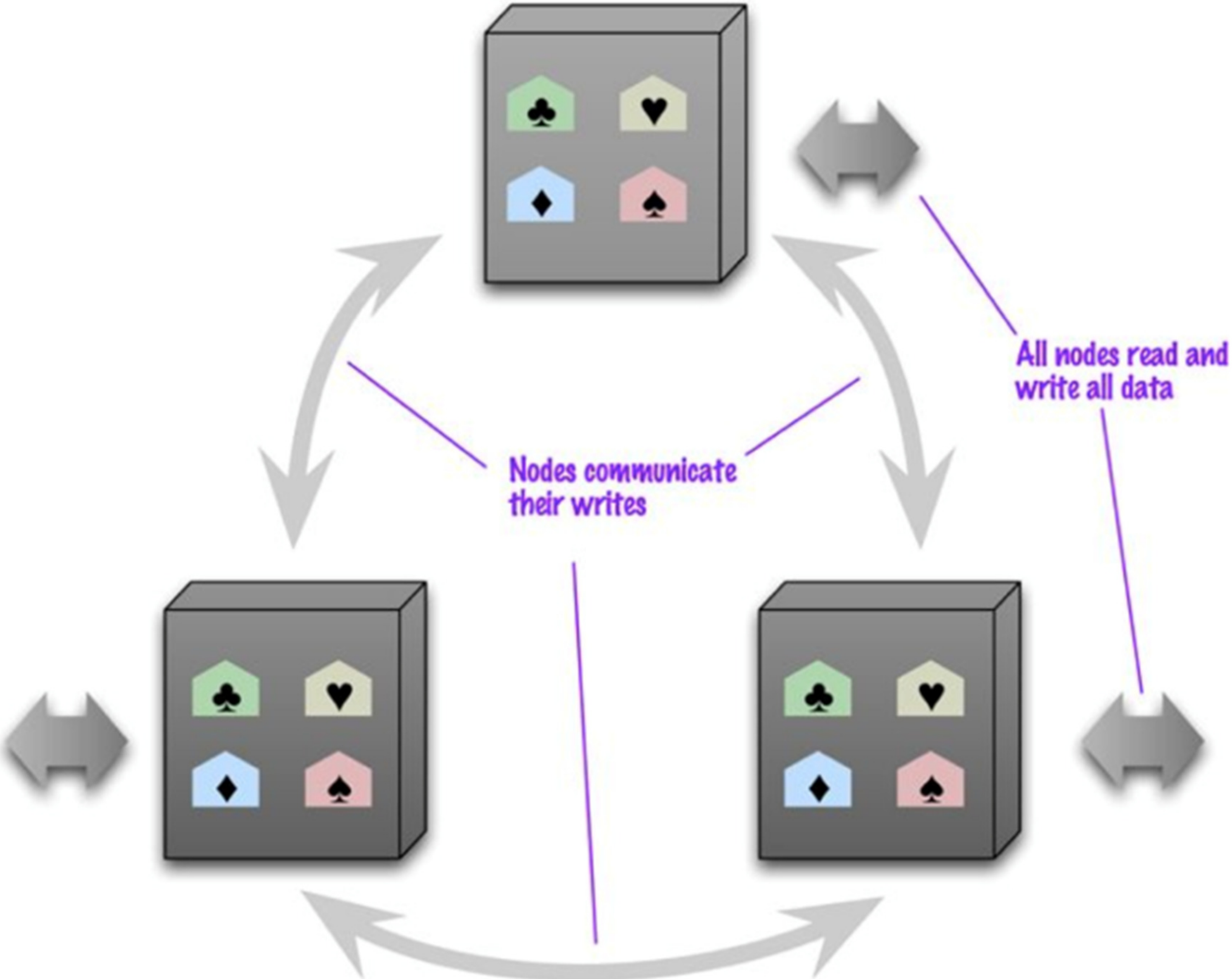
- Master-slave replication is most helpful for scaling when you have a read-intensive dataset, since read accesses can be on any node, but it isn't such a good scheme for datasets with heavy write traffic, since all writes must be routed to the master and than propagated to the slaves.
- Also, should the master fail, the slaves can still handle read requests (read resilience)
- The failure of the master however eliminates the ability to handle writes until either the master is restored or a new master is appointed. Having slaves as replicates of the master does speed up recovery after a failure
- The main drawback is the arising of possible inconsistency. You have the danger that different clients, reading different slaves, will see different values because the changes haven't all propagated to the slaves!

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

- 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 does not prevent access to the data store

Peer-to-peer replication

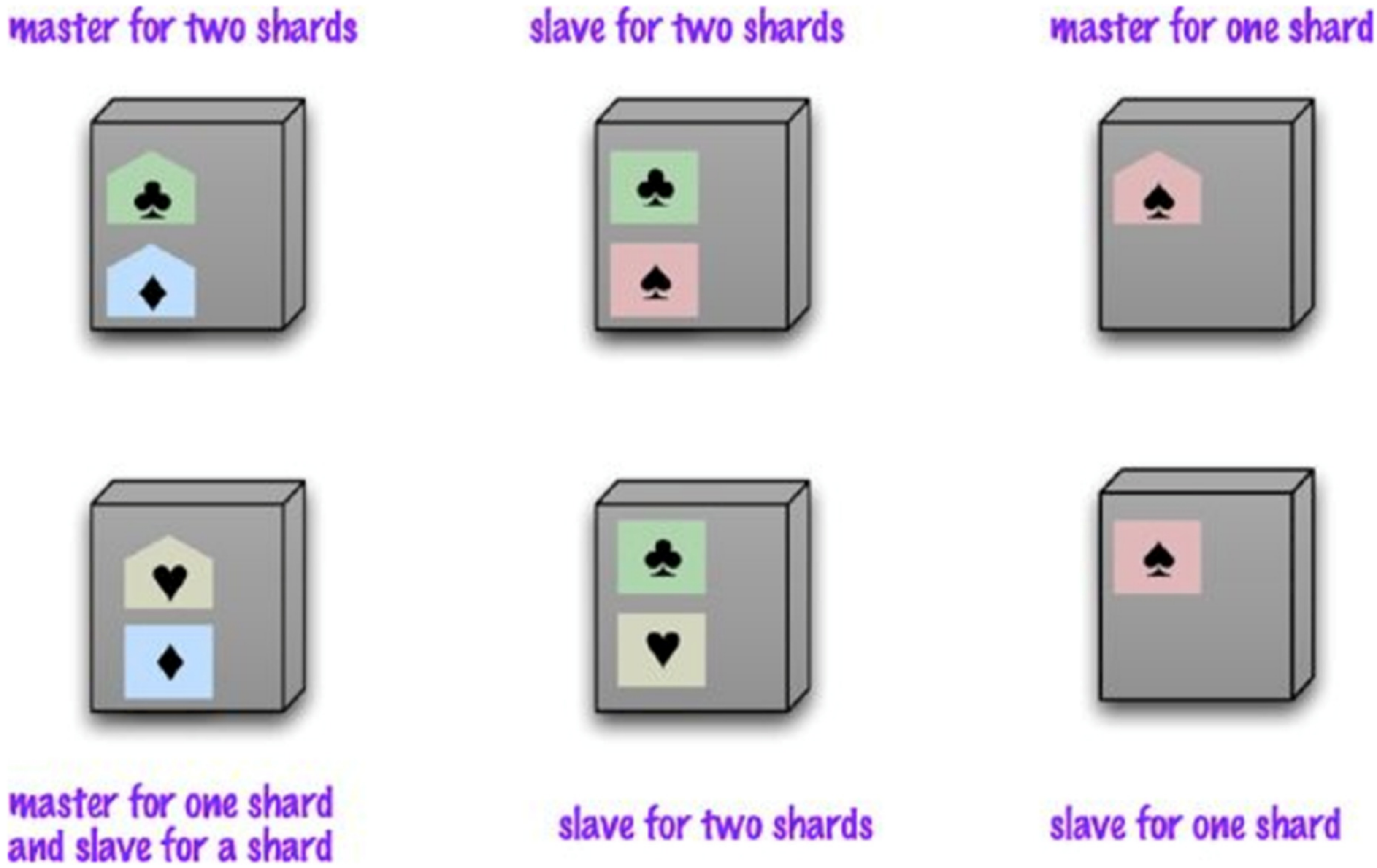


Peer-to-peer replication

- With a peer-to-peer replication cluster, you can ride over node failures without losing access to data. Furthermore, you can easily add nodes to improve your performance
- The biggest complication is, again, consistency. When you can write to two different places, you run the risk that two people will attempt to update the same record at the same time: a write-write conflict!
- Inconsistencies on read lead to problems but at least they are relatively transient. Inconsistent writes are forever!
- There are various policies that can be adopted to cope with this problem. Here we mention only some of them:
  - we can ensure that whenever we write data, the replicas coordinate one another to ensure we avoid a conflict (at the cost of network traffic to coordinate the writes)
  - In other cases we can decide we don't need all the replicas to agree on the write, but just a majority
  - At the other extreme, we can decide to cope with an inconsistent write. There are contexts when we can come up with policy to successively merge inconsistent writes

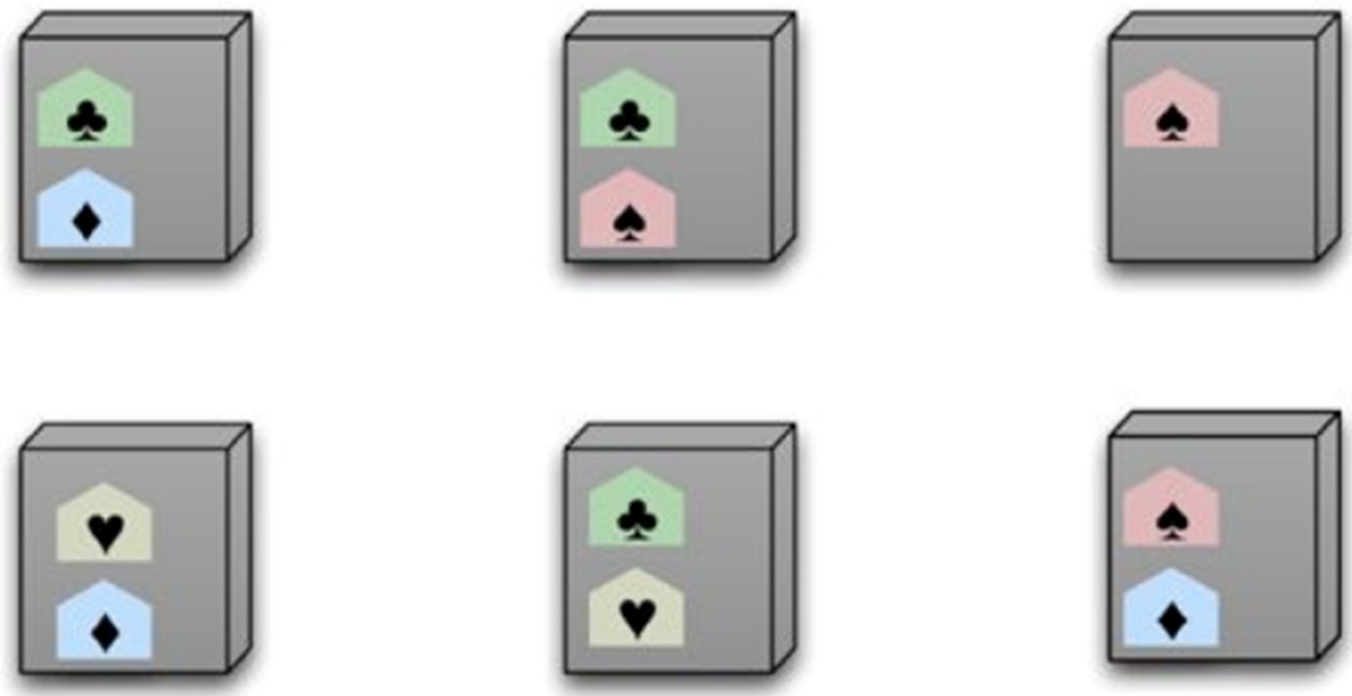
Sharding with Master-slave replication

- If we use both master-slave replication and sharding, we then have multiple masters, but each data item only has a single master.



Sharding with P2P replication

- Using peer-to-peer replication and sharding is a common strategy for column-family databases. Each shard is replicated in a peer-to-peer fashion
- In a scenario like this you might have tens or hundreds of nodes in a cluster with data sharded over them. Should a node fail, then the shards on that node will be built on the other nodes



NoSQL databases: Aggregated DBs

- NoSQL data models
- Key-value, document, column databases
- Distribution models
- Consistency
- Map-Reduce

Consistency

- Informally: "every request receives the right response"
- In relational databases the above aim is achieved through the enforcement of so-called **ACID** properties to database transactions (**strong consistency**)
  - Atomicity: every transaction is executed in "all-or-nothing" fashion
  - Coherence*: every transaction preserves the coherence with constraints on data (i.e., at the end of the transaction constraints are satisfied by data)
  - Isolation: transaction does not interfere. Every transaction is executed as it was the only one in the system (every seralization of concurrent transactions is accepted)
  - Durability: after a commit, the updates made are permanent regardless possible failures

***Note**: In DB literature the term Consistency is often used in place of Coherence to indicate conformance of data with integrity constraints. However, here we use Consistency in a wider sense (e.g., it indicates the enforcement of all ACID properties), and thus we prefer to indicate constraints satisfaction as data Coherence.

Consistency

Transactions in relational DBMSs are such that:

- Atomicity is always guaranteed
- Coherence can be relaxed within the transaction (e.g., by deferring constraint checks), but it is enforced at the end of the transaction (when data are committed)
- Durability is in general guaranteed (even though some in-memory databases may not fully meet this requirement)
- Isolation is the most critical property, since independent transactions may interfere in various ways.

Consistency – Isolation in RDBMSs

- Possible Conflicts between transactions T1 and T2 are classified as
  - **Write-Read (WR)**: T1 reads data written by T2, which is still not committed (and that could be rolled-back)
  - **Read-Write (RW)**: T1 reads data that T2 then updates (before T1 is committed)
  - **Write-Write (WW)**: T2 overwrites data previously written by T1 (but before T1 is committed)
- Isolation is ensured through the use of sophisticated lock mechanisms that assign a resource to a transaction in exclusive or shared modality
- DBMSs allow users to set the desired isolation level, which means adopting a desired lock policy
- The stricter lock policy guarantees serializability: the final effect on the DB is the execution of either the sequence T1-T2 or the sequence T2-T1
- For performance reasons, often a weaker isolation level is adopted: for example, the Read-Committed level is enough to avoid WW and WR conflicts

RW - example

In this example, we want to check whether X+Y+Z=100

Tempo	T1	T2	X	Y	Z
t1	beginTrans		20	30	50
t2	read(X)	beginTrans	20	30	50
t3	read(Y)	read(Y)	20	30	50
t4		Y := Y-10	20	30	50
t5		read(Z)	20	30	50
t6		Z := Z+10	20	30	50
t7		write(Y)	20	20	50
t8		write(Z)	20	20	60
t9	read(Z)	commit	20	20	60
t10	s=X+Y+Z		20	20	60
t11	commit		20	20	60

This kind of conflicts is also called inconsistent read. Avoiding them means ensuring **logical consistency**

Consistency in NoSQL DBs

- A common claim we hear is that NoSQL databases don’t support transactions and thus can’t be consistent
- As we already said, any statement about lack of transactions usually only applies to some NoSQL databases, in particular the aggregate-oriented ones, whereas graph databases tend to support ACID transactions
- Secondly, aggregate-oriented databases do support atomic operations, but only within a single aggregate. Consider the example in the previous slide and assume you have an aggregate database storing that data. If X, Y and Z are managed in the same aggregate, RW conflict can be avoided (logical consistency within an aggregate but not between aggregates)

Relaxing consistency

- More in general, in distributed databases, consistency, and in particular isolation, is often relaxed for performance reasons in such a way that also conflicts typically avoided in a single-server setting (like WW conflicts) are somehow tolerated

- Indeed, concurrent programming involves a fundamental tradeoff between safety (avoiding errors such as WW conflicts) and liveness (responding quickly to clients)

- Also, we notice that in the presence of more than one server, such as with peer-to-peer replication, serializability is complicated by the fact that different nodes might apply the updates in a different order. Often, when people talk about concurrency in distributed systems, they talk about sequential consistency— ensuring that all nodes apply operations in the same order.

Update consistency

- As for updates, rather than a **pessimistic approach**, which works by preventing conflicts from occurring, an **optimistic approach** is often adopted, which lets conflicts occur, but detects them and takes action to sort them out
- A common optimistic approach is a **conditional update** where any client that does an update (which may give rise to WW conflicts) tests the value just before updating it to see if it's changed since her last read
- There is another optimistic way to handle a write-write conflict — save both updates and record that they are in conflict. Then, you have to merge the two updates somehow (heavily application dependent)
- These techniques are particularly relevant under P2P replication

Read consistency

- Since in NoSQL databases logical consistency is within an aggregate but not between aggregates, 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**

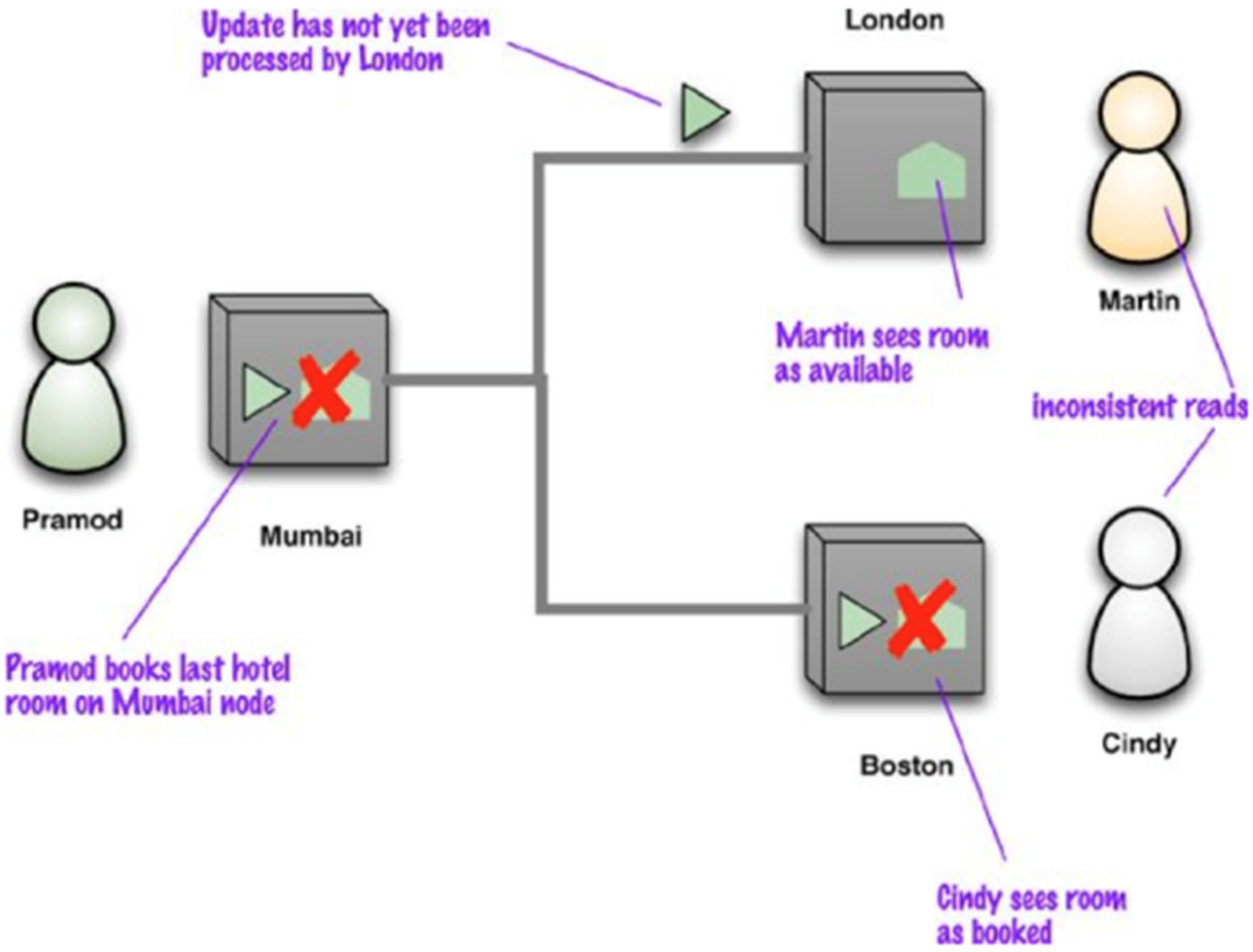
- A NoSQL system may have a quite short inconsistency window (e.g., Amazon's documentation says that the inconsistency window for its SimpleDB service is usually less than a second)

- Once you introduce replication, however, you **get a whole new kind of read inconsistency** (besides logical inconsistency): Replication inconsistency

Replication consistency

Replication consistency amounts to ensuring that the same data item has the same value when read from different replicas

Example: Martin and Cindy, who are in London and Boston respectively, are on phone to book together an hotel room. At the same time, Pramod books the last room in that hotel at the Mumbai node and the Boston node shows the booking before the London one



Eventually consistent

- Eventually, of course, the updates propagate fully in the network, (and Martin will see the room is booked)
- Therefore this situation is generally referred to as **eventually consistent**, meaning that at any time nodes may have replication inconsistencies but, if there are no further updates, eventually all nodes will be updated to the same value
- Although replication consistency is independent from logical consistency, replication can exacerbate a logical inconsistency by lengthening its inconsistency window (for examples, under master-slave replication, inconsistency window is generally narrow on the master, but lasts for much longer on a slave)

Session consistency

- Inconsistency windows can be particularly problematic when you get inconsistencies with yourself.
- **read-your-writes consistency** means that, once you've made an update, you're guaranteed to continue seeing that update.
- There are situations where guaranteeing this may not be so obvious
- Consider the example of posting comments on a blog entry. Often, systems handle the load of such sites by running on a cluster and load-balancing incoming requests to different nodes. This means that 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!
- **session consistency** guarantees **read-your-writes consistency** within a session: if the session ends for some reason, or the user access simultaneously the same system from different computers, she may lose consistency.

Session consistency

- There are a couple of techniques to guarantee session consistency.

  - **sticky session**: a session that's tied to one node (this is also called session affinity). The downside is that sticky sessions reduce the ability of the load balancer to do its job.

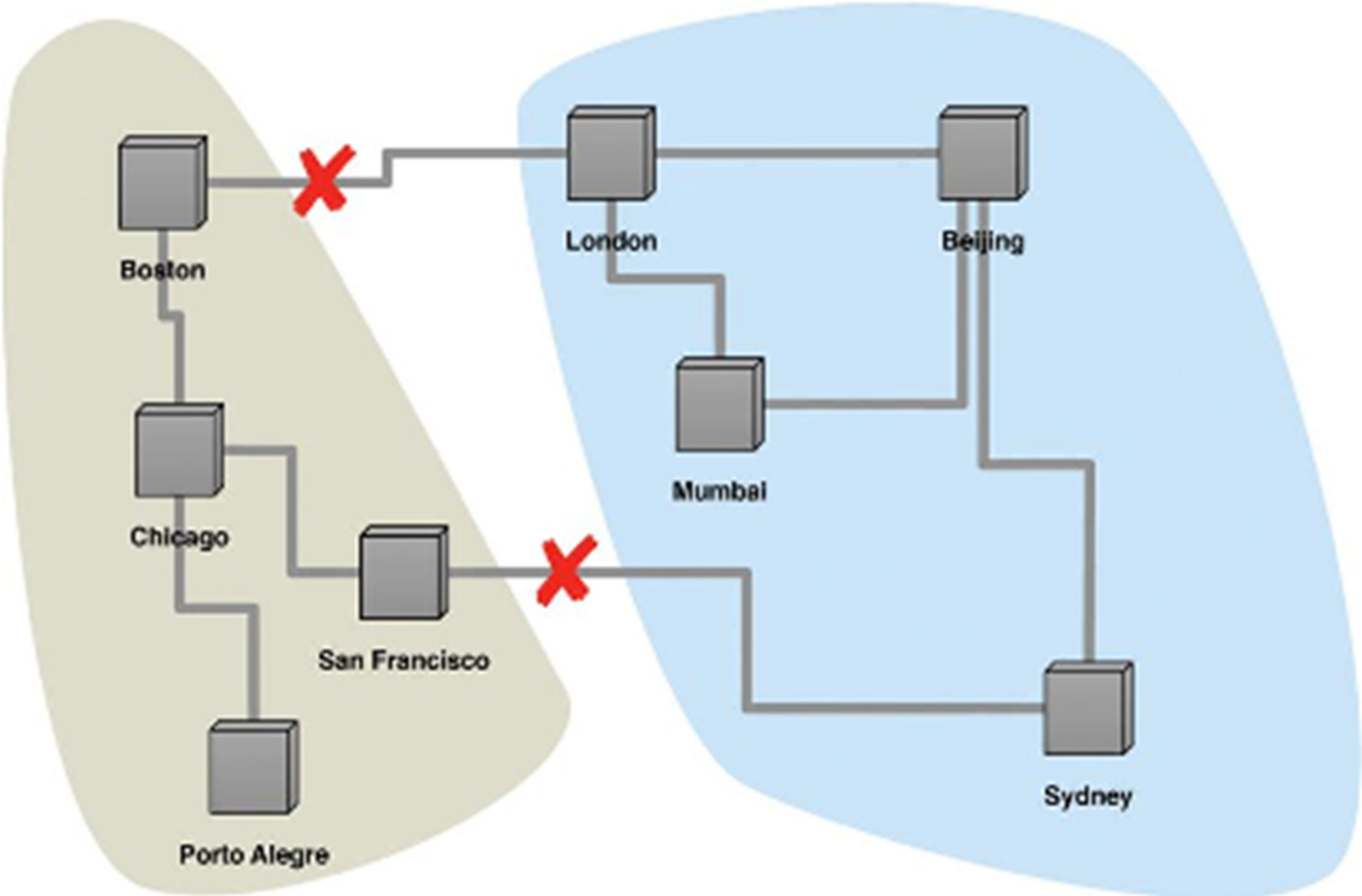
  - **using version stamps*,** ensuring that every interaction with the data store includes the latest version stamp seen by a session. The server node must then ensure that it has the updates that include that version stamp before responding to a request.

*A version stamp is a field that changes every time the data in the database changes. When you read the data you keep a note of the version stamp, so that when you write data you can check to see if the version has changed.

Properties of shared data systems

- Consistency
  - ✓ (informally) "every request receives the right response"
  - ✓ E.g. If I get my shopping list on Amazon I expect it contains all the previously selected items
  - ✓ It is essentially what discussed so far
- Availability
  - ✓ (informally) "each request eventually receives a response"
  - ✓ E.g. eventually I access my shopping list
  - ✓ A bit more formally: **if you can talk to a node in the cluster, it can read and write data**.
- tolerance to network Partitions
  - ✓ (informally) "servers can be partitioned into multiple groups that cannot communicate with one other"
  - ✓ A bit more formally: the cluster can survive communication breakages that separate the cluster into multiple partitions unable to communicate one another

Network partitions



This situation is also known as a **split brain**

(simple) CA systems

- A single-server system is the obvious example of a CA system: a system that has Consistency and Availability. Partition tolerance is not applicable here, since a single machine can't partition. This is the world that most relational database systems live in

- It is theoretically possible to have a CA cluster: However, this would mean that if a partition ever occurs in the cluster, all the nodes in the cluster would go down so that no client can talk to a node (according to definition of availability, if a node is not reachable, then it does not infer lack of capability). Realizing a CA cluster in this way is usually prohibitively expensive. Notice also that we are somehow bypassing partition tolerance here

The CAP Theorem

- When we try to have a cluster to be really tolerant to network partitions, the CAP theorem comes into play

"Of three properties of shared-data systems (Consistency, Availability and tolerance to network Partitions) only two can be achieved at any given moment in time."

- 2000: Eric Brewer, PODC conference keynote
- 2002: Seth Gilbert and Nancy Lynch, ACM SIGACT News 33(2)

The CAP Theorem - observations

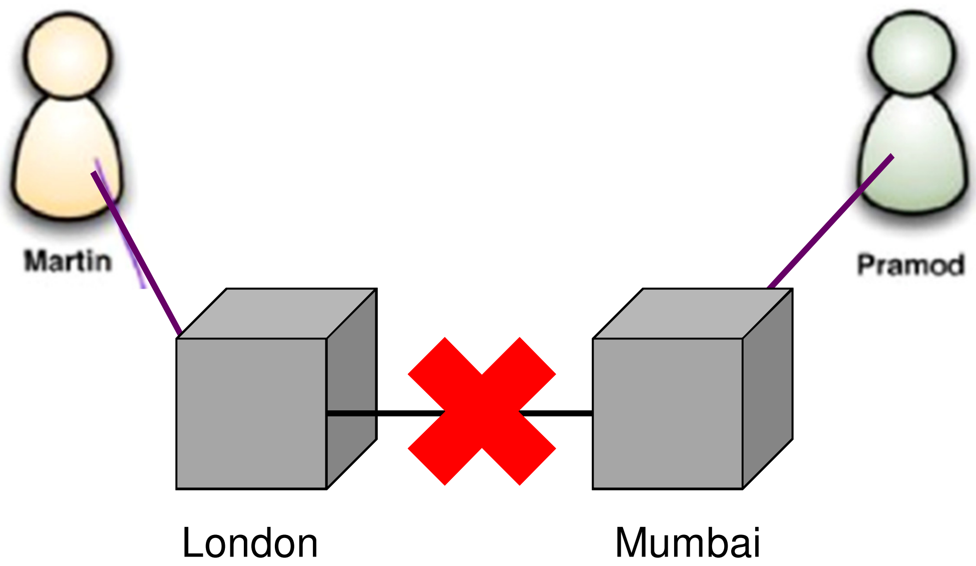
- CAP states that in case of failures you can have at most two of these three properties for any shared-data system

- To scale out, you have to distribute resources.
  - ✓ P is not really an option but rather a need
  - ✓ The real selection is among Consistency or Availability
  - ✓ In almost all cases, you would choose availability over consistency.

- In practice, what CAP is saying is that in a system that may suffer partitions, as distributed system do, you have to trade off consistency versus availability (if there are no strong reasons to do the converse). However, it is not a binary decision; often, you can trade off a little consistency to get some availability.

Example

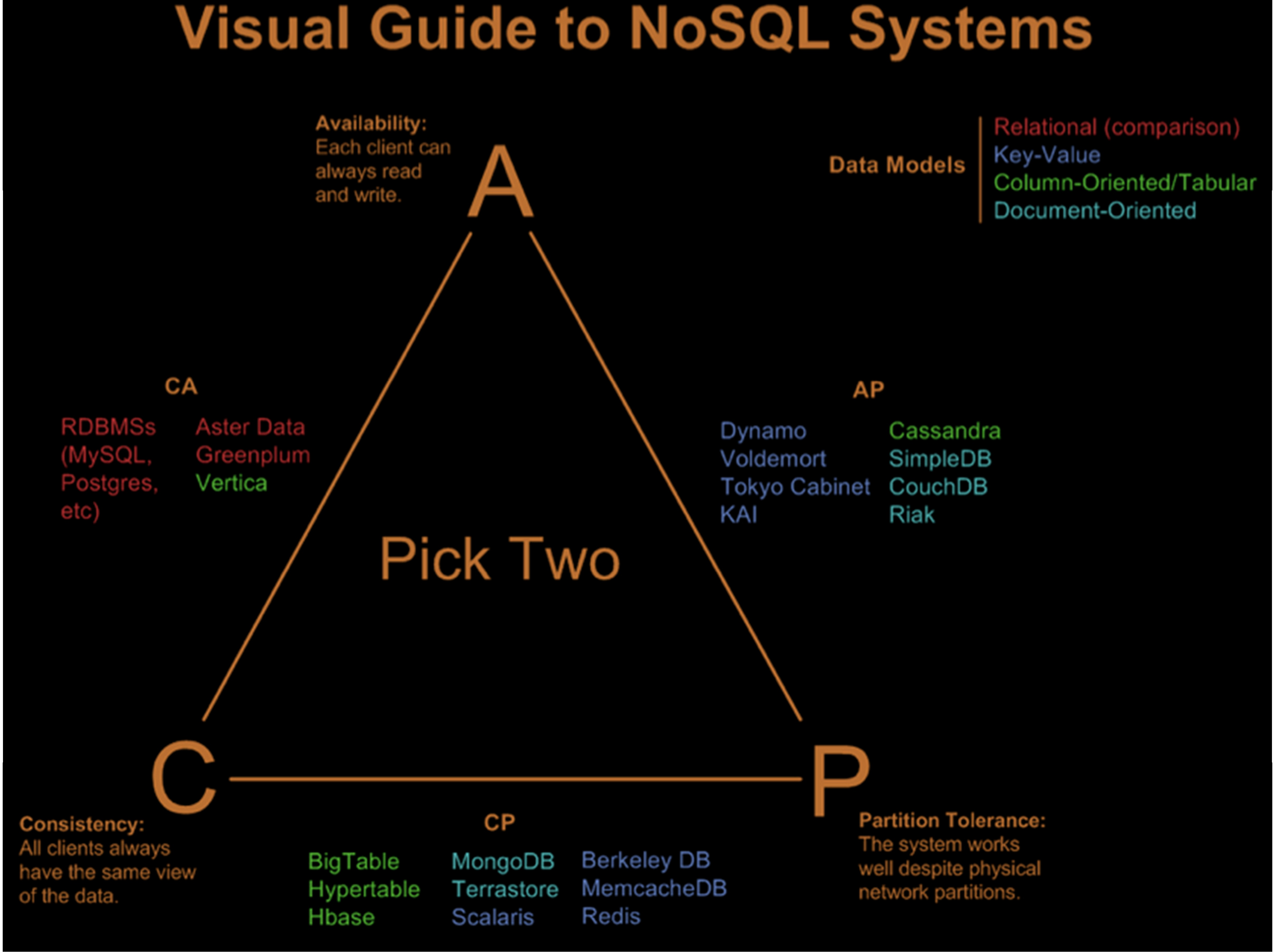
Martin and Pramod are both trying to book the last hotel room on a system that uses peer-to-peer distribution with two nodes. If we want to ensure consistency, when Martin tries to book his room on the London node, that node must communicate with the Mumbai node before confirming the booking. But should the network link break, then neither system can book any hotel room, sacrificing availability.



Example

- One way to improve availability is to designate one node as the master for a particular hotel and ensure all bookings are processed by that master. Should that master be Mumbai, then Mumbai can still process hotel bookings for that hotel and Pramod will get the last room, whereas Martin can see the inconsistent room information but cannot make a booking (which would in this case cause an update inconsistency). This is a lack of availability for Martin.

- To gain more availability, we might allow both systems to keep accepting hotel reservations even if a link in the network breaks down. But this may cause both Martin a Pramod book the same room => Inconsistency. But in this domain it might be tolerated somehow: the travel company may tolerate some overbooking; some hotels might always keep a few rooms clear even when they are fully booked; Some hotels might even cancel the booking with an apology once they detected the conflict.


Visual Guide to NoSQL Systems
Availability:
Each client can always read and write.
A
Data Models
Relational (comparison)
Key-Value
Column-Oriented/Tabular
Document-Oriented
CA
RDBMSs (MySQL, Postgres, etc)
Aster Data
Greenplum
Vertica
AP
Dynamo
Voldemort
Tokyo Cabinet
KAI
Cassandra
SimpleDB
CouchDB
Riak
Pick Two
C
P
Consistency:
All clients always have the same view of the data.
CP
BigTable
Hypertable
Hbase
MongoDB
Terrastore
Scalaris
Berkeley DB
MemcacheDB
Redis
Partition Tolerance:
The system works well despite physical network partitions.

Relaxing consistency

- The lesson here is that there are cases in which you can gracefully deal with inconsistent answers to requests

- The level of inconsistency accepted completely depend on the application at hand. In general, you should establish how tolerant you are to **stale reads** (i.e., reading obsolete data) and how long the **inconsistency window** can be

- Advocate of NoSQL often say that instead of following the ACID properties of relational transactions, NoSQL systems follow the **BASE** properties

BASE

BASE stands for Basically Available Soft State Eventually Consistent system.

- Basically Available: the system is available most of the time and there could exist a subsystem temporarily unavailable
- Soft State: data are “volatile” in the sense that their persistence is in the hand of the user that must take care of refreshing them
- Eventually Consistent: the system eventually converges to a consistent state

CAP theorem - conclusions

- It’s usually better to think not about the tradeoff between consistency and availability but rather between consistency and latency

- We can always improve consistency by getting more nodes involved in the interaction, but each node we add increases the response time of that interaction

- We can then think of availability as the limit of latency that we’re prepared to tolerate: once latency gets too high, we give up and treat the data as unavailable

- In principle every system should be designed to ensure both C and A in normal situations. When a partition occurs, the decision among C and A can be taken. When the partition is resolved the system takes corrective action coming back to work in normal situation

NoSQL databases: Aggregated DBs

- NoSQL data models
- Key-value, document, column databases
- Distribution models
- Consistency
- Map-Reduce

Map Reduce

- When you have a cluster, you have lots of machines to spread the computation over.

- However, you also still need to try to reduce the amount of data that needs to be transferred across the network.

- The map-reduce pattern is a way to organize processing in such a way as to take advantage of multiple machine on a cluster while keeping as much processing and the data it needs together on the same machine.

Map Reduce

- This programming model gained prominence with Google's MapReduce framework [Dean and Ghemawat, OSDI-04].
- A widely used open-source implementation is part of the Apache Hadoop project.
- The name "map-reduce" reveals its inspiration from the map and reduce operations on collections in functional programming languages.

Map Reduce - benefits

- Complex details are abstracted away from the developer
  - No file I/O
  - No networking code
  - No synchronization
- It's scalable because you process one record at a time
- A record consists of a key and corresponding value

Map Reduce Example* – Job input

- Each mapper gets a chunk of job's input data to process
  - This "chunk" is called an **InputSplit**
- In this example the input is a portion of a log with a list of events, each of a certain type (INFO, WARN…)

```
2012-09-06 22:16:49.391 CDT INFO "This can wait"
2012-09-06 22:16:49.392 CDT INFO "Blah blah"
2012-09-06 22:16:49.394 CDT WARN "Hmmm..."
2012-09-06 22:16:49.395 CDT INFO "More blather"
2012-09-06 22:16:49.397 CDT WARN "Hey there"
2012-09-06 22:16:49.398 CDT INFO "Spewing data"
2012-09-06 22:16:49.399 CDT ERROR "Oh boy!"
```

*This example is taken from: An Introduction to Hadoop. Mark Fei (Cloudera). Strata + Hadoop World 2012 Conference

Map Reduce Example – Phyton code for map function

- Our map function will parse the event type, and then output that event (key) and a literal 1 (value)

```
#!/usr/bin/env python

import sys

levels = ['TRACE', 'DEBUG', 'INFO',
          'WARN', 'ERROR', 'FATAL']

for line in sys.stdin:
    fields = line.split()
    for field in fields:
        field = field.strip().upper()
        if field in levels:
            print "%s\t1" % field
```

Boilerplate Python stuff

Define list of JUnit log levels

Split every line (record) we receive on standard input into fields, normalized by case

If this field matches a log level, print it (and a 1)

Map Reduce Example– output of map function

- The map function produces key/value pairs as output

```
INFO     1
INFO     1
WARN     1
INFO     1
WARN     1
INFO     1
ERROR    1
```

Map Reduce Example –
Input to Reduce Function

- The (single) Reducer receives a key and all values for that key
- Keys are always passed to reducers in sorted order whereas, values are unordered

ERROR	1
INFO	1
INFO	1
INFO	1
INFO	1
WARN	1
WARN	1

Map Reduce Example –
Python Code for Reduce Function

- The Reducer first extracts the key and value it was passed

```
#!/usr/bin/env python

import sys

previous_key = ''
sum = 0

for line in sys.stdin:
    fields = line.split()
    key, value = line.split()

    value = int(value)
    # continued on next slide
```

Boilerplate Python stuff

Initialize loop variables

Extract the key and value passed via standard input

Map Reduce Example –
Python Code for Reduce Function

- Then simply adds up the value for each key

```
    # continued from previous slide
    if key == previous_key:
        sum = sum + value
    else:
        if  previous_key != '':
            print '%s\t%i' % (previous_key, sum)
        previous_key = key
        sum = 1

print '%s\t%i' % (previous_key, sum)
```

If key unchanged, increment the count

If key changed, print sum for previous key

Re-init loop variables

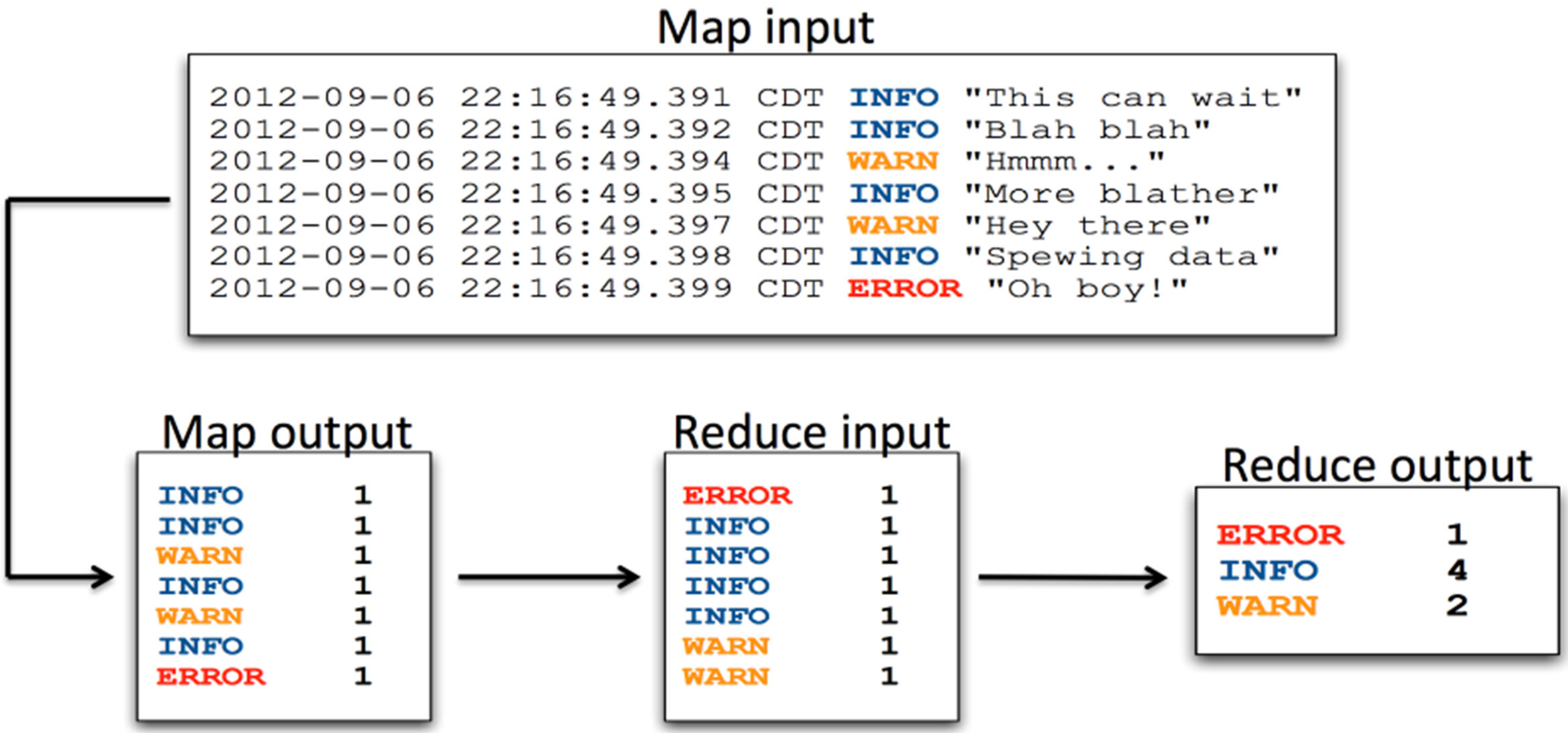
Print sum for final key

Map Reduce Example –
Output of the Reduce Function

- The output of this Reduce function is a sum for each level

ERROR	1
INFO	4
WARN	2

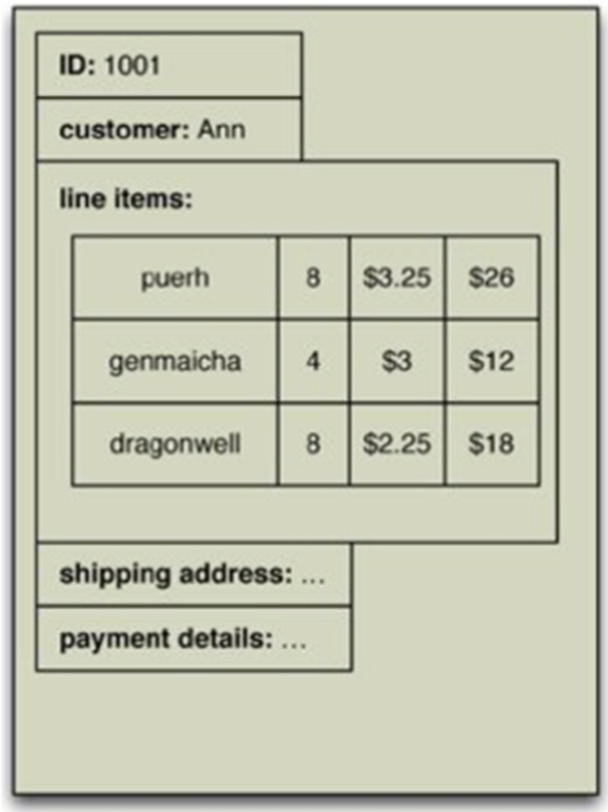
Map Reduce Example – Recap in data flow



An Example with Aggregate data

Let us consider the usual scenario of customers and orders

We have chosen order as our aggregate, with each order having line items. Each line item has a product ID, quantity, and the price charged.



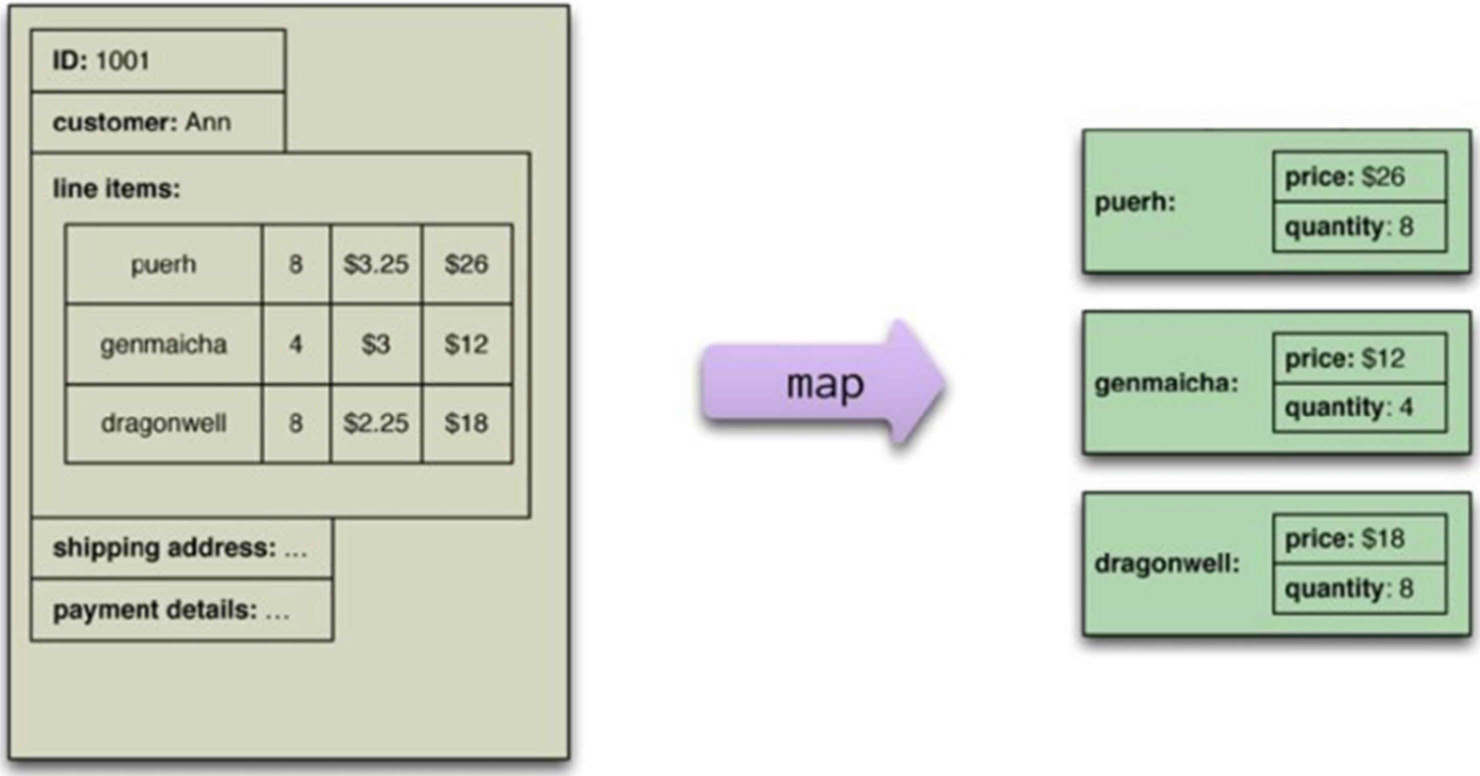
Sales analysis people want to see a product and its total revenue for the last seven days.

An Example with Aggregate data

- In order to get the product revenue report, you'll have to visit every machine in the cluster and examine many records on each machine.
- This is exactly the kind of situation that calls for map-reduce. Again, the first stage in a map-reduce job is the map.
- A map is a function whose input is a single aggregate and whose output is a bunch of key-value pairs.
- In this case, the input would be an order, and the output would be key-value pairs corresponding to the line items
- For this example, we are just selecting a value out of the record, but there's no reason why we can't carry out some arbitrarily complex function as part of the map—providing it only depends on one aggregate's worth of data.

An Example with Aggregate data

Each such pair would have the product ID as the key and an embedded map with the quantity and price as the value



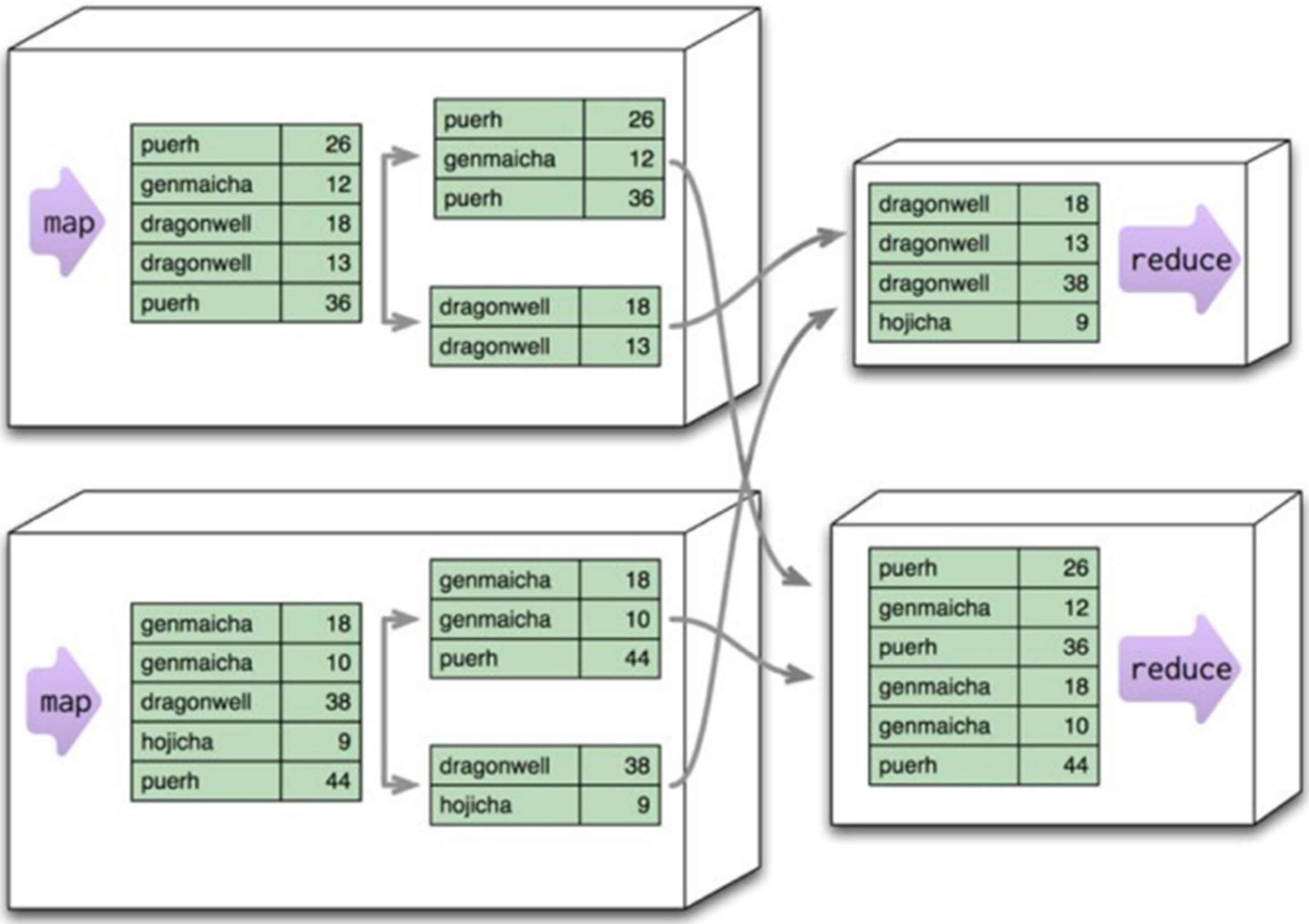
An Example with Aggregate data

The reduce function takes multiple map outputs with the same key and combines their values

An Example with Aggregate data

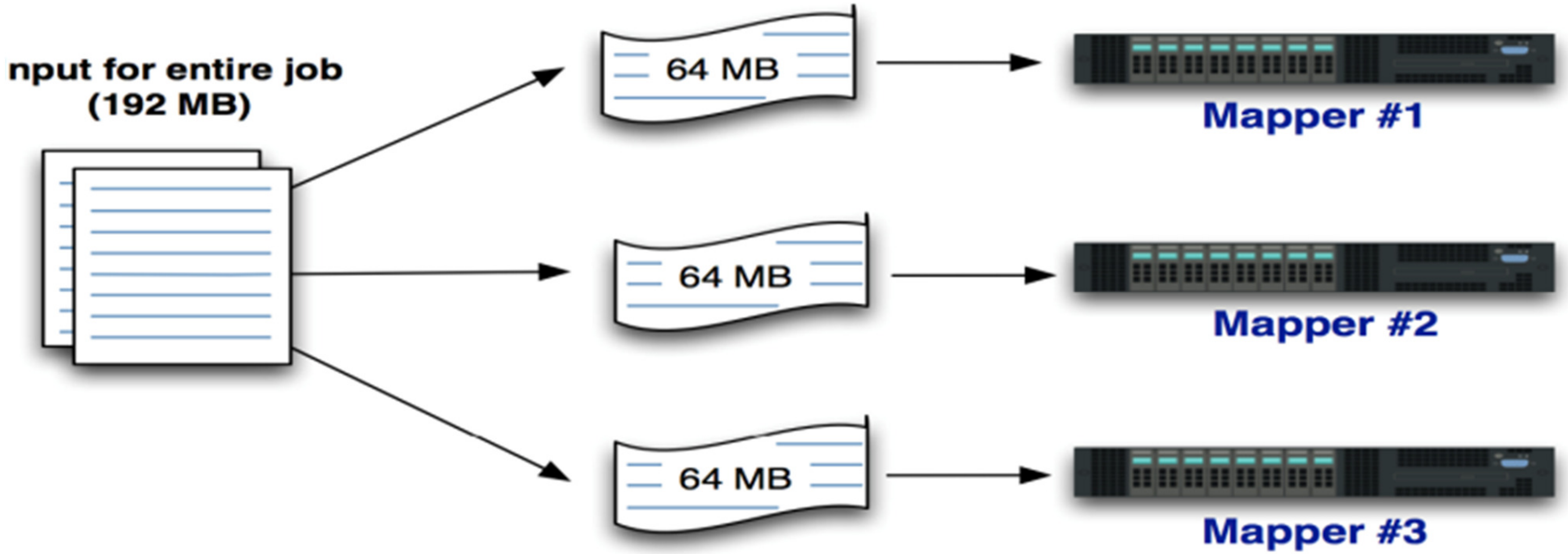
- The map-reduce framework arranges for map tasks to be run on the correct nodes to process all the documents and for data to be moved to the reduce framework
- The framework collects all the values for a single pair and calls the reduce function once with the key and the collection of all the values for that key
- So to run a map-reduce job, you just need to write these two functions.
- Each application of the map function is independent of all the others. This allows them to be safely parallelizable, so that a map-reduce framework can create efficient map tasks on each node and freely allocate each order to a map task.
- To increase parallelism, we can also partition the output of the mappers and send each partition to a different reducer ("**shuffling**")

Partitioning map outputs



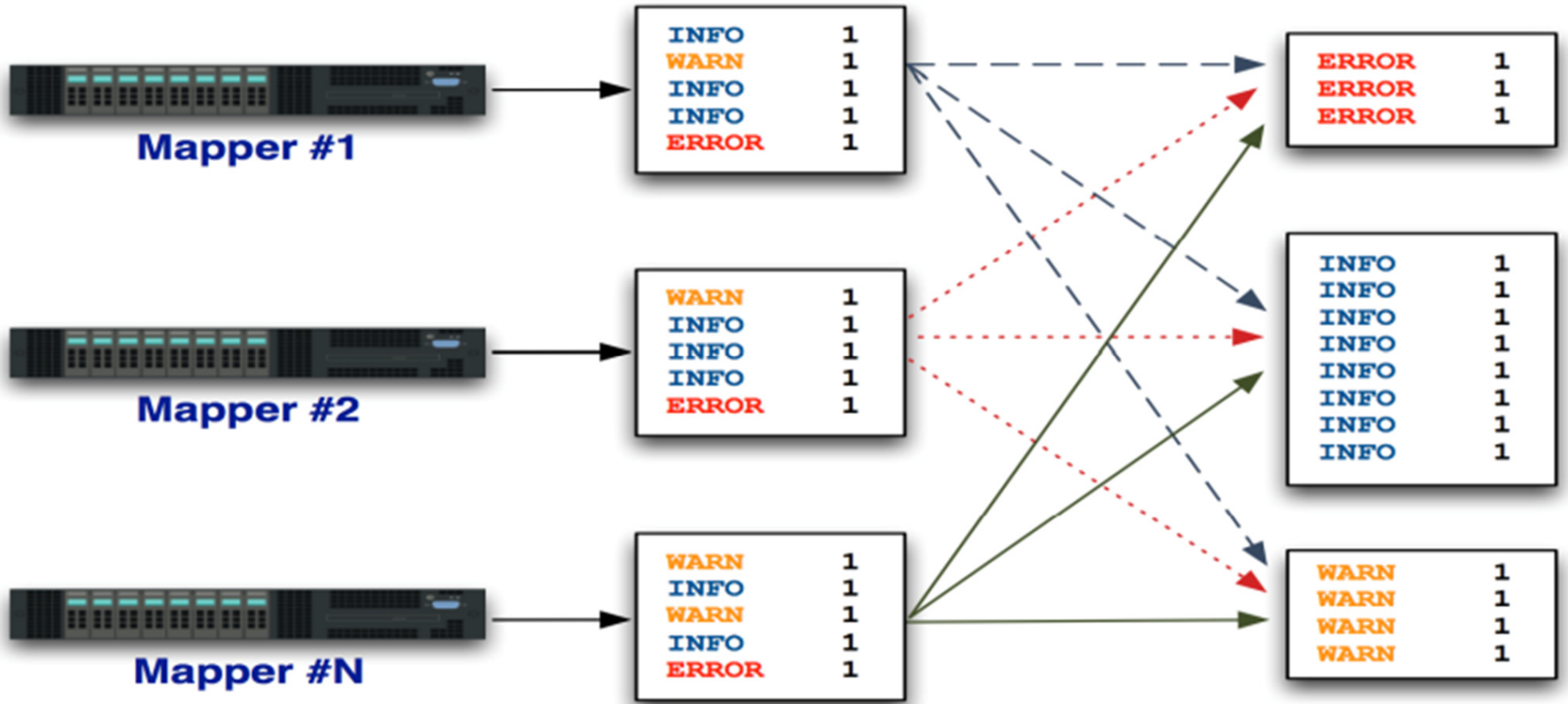
Looking at the entire distributed computation

- Coming back to the log example we might have the following situation:
  - Input for the entire job is subdivided into InputSplits
    - In Hadoop An InputSplit *usually* corresponds to a single HDFS (Hadoop Distributed File System) block
    - Each of these serves as input to a single Map task



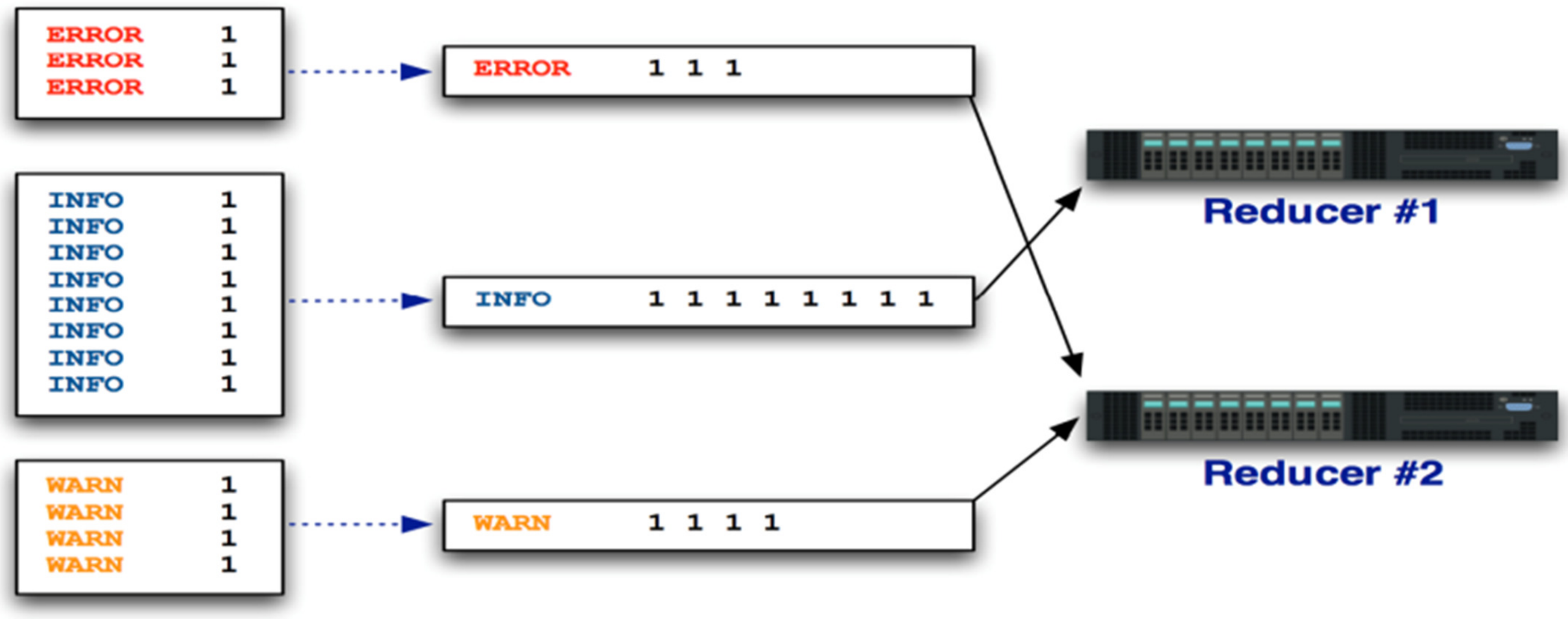
Mappers Feed the Shuffle and Sort

Output of **all** Mappers is partioned, merged, and sorted (No code required – the framework, e.g., Hadoop does this automatically)

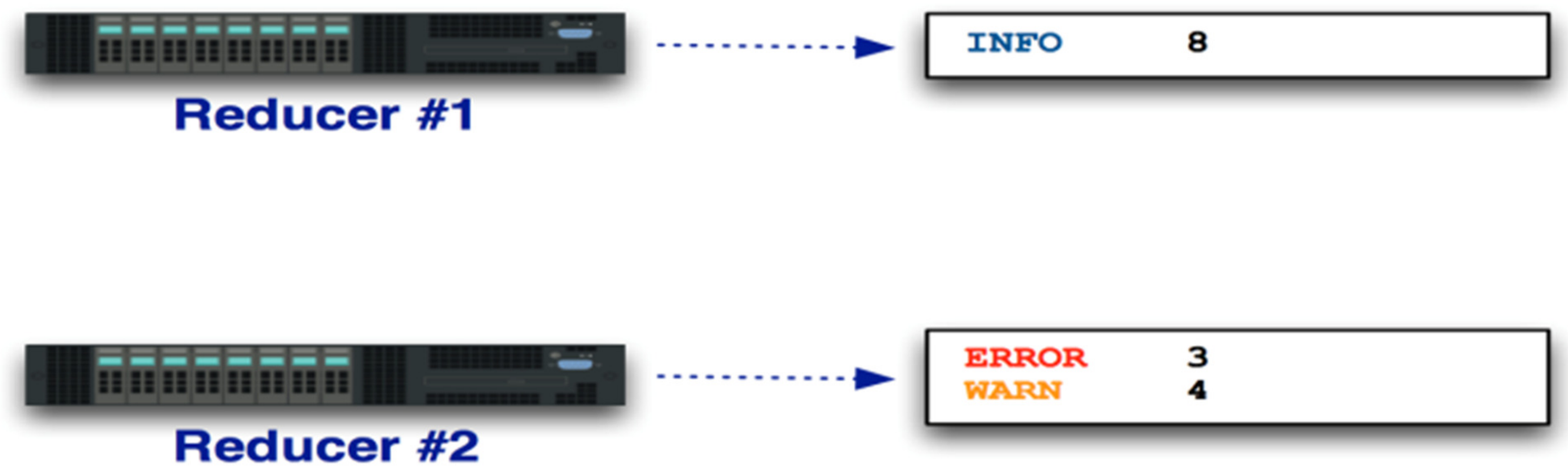


Shuffle and Sort Feeds the Reducers

All values for a given key are then collapsed into a list . The key and all its values are fed to reducers as input



Each Reducer Has an Output



In Hadoop, these are output files stored in HDFS below your output directory. It is then possible to replicate them to a local copy

Programming with map-reduce

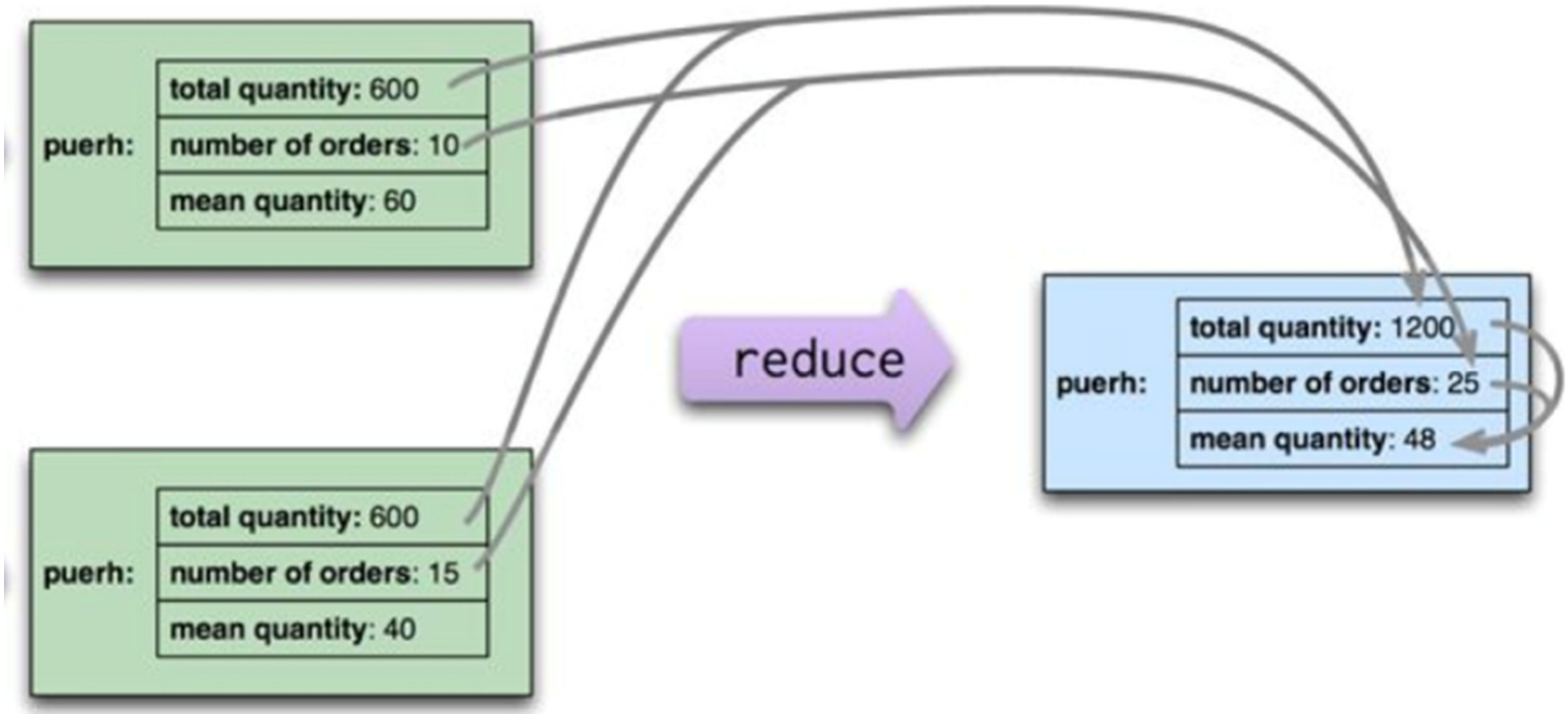
- Map-reduce is powerful, but it has a rigid schema: Within a map task, you can only operate on a single aggregate. Within a reduce task, you can only operate on a single key.

- This means you have to think differently about structuring your programs so they work well within these constraints. In particular, you have to structure your calculations around operations that fit in well with the notion of a reduce operation.

- Of course, you have to put some care in this

A further example

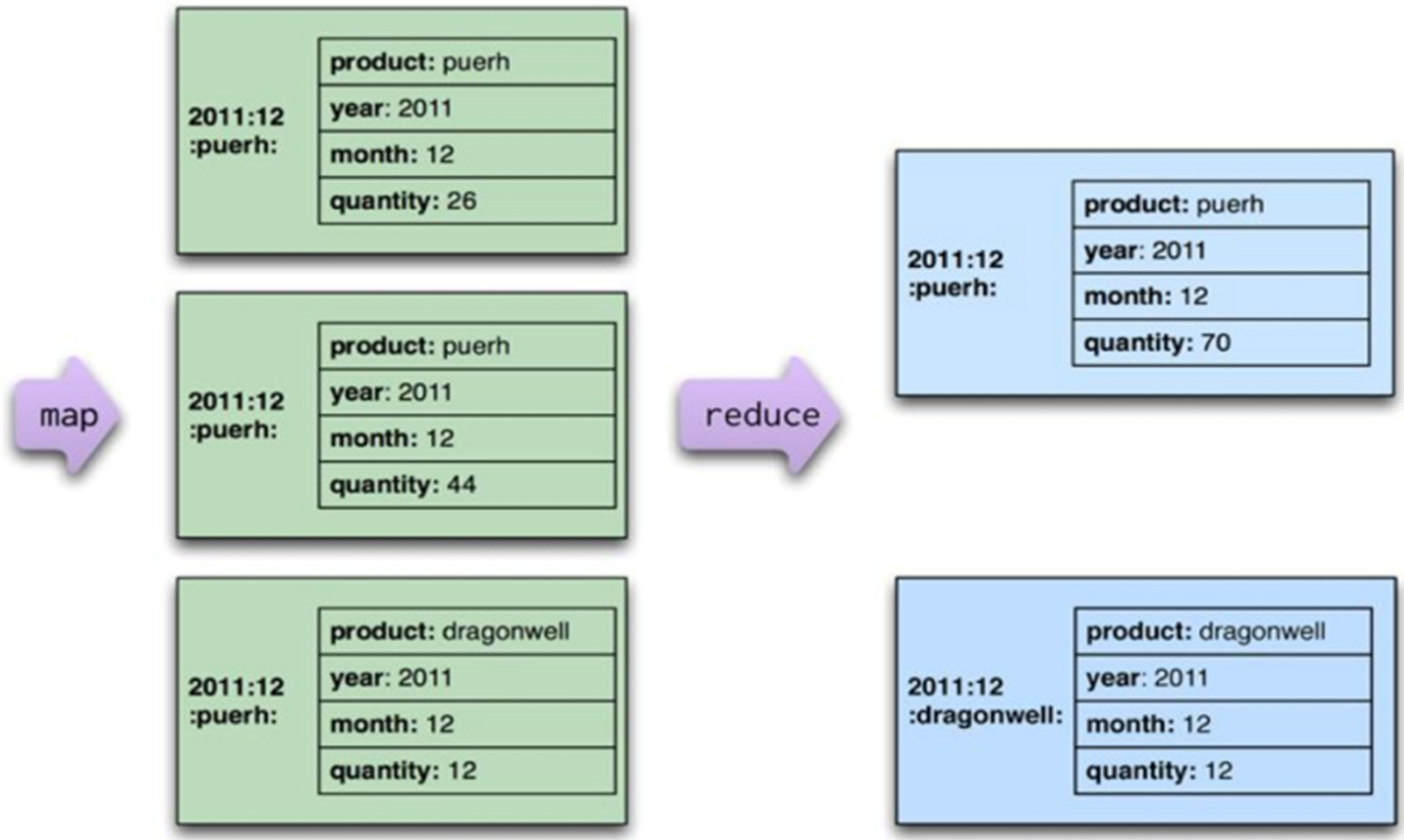
- Let's consider the kind of orders we've been looking at so far; suppose we want to know the average ordered quantity of each product.
- But averages are not composable — that is, if I take two groups of orders, I can't combine their averages alone.
- In this case, the reducer needs to take total amount and the count of orders from each group , combine those, and then calculate the average from the combined sum and count



A two stage Map-Reduce example

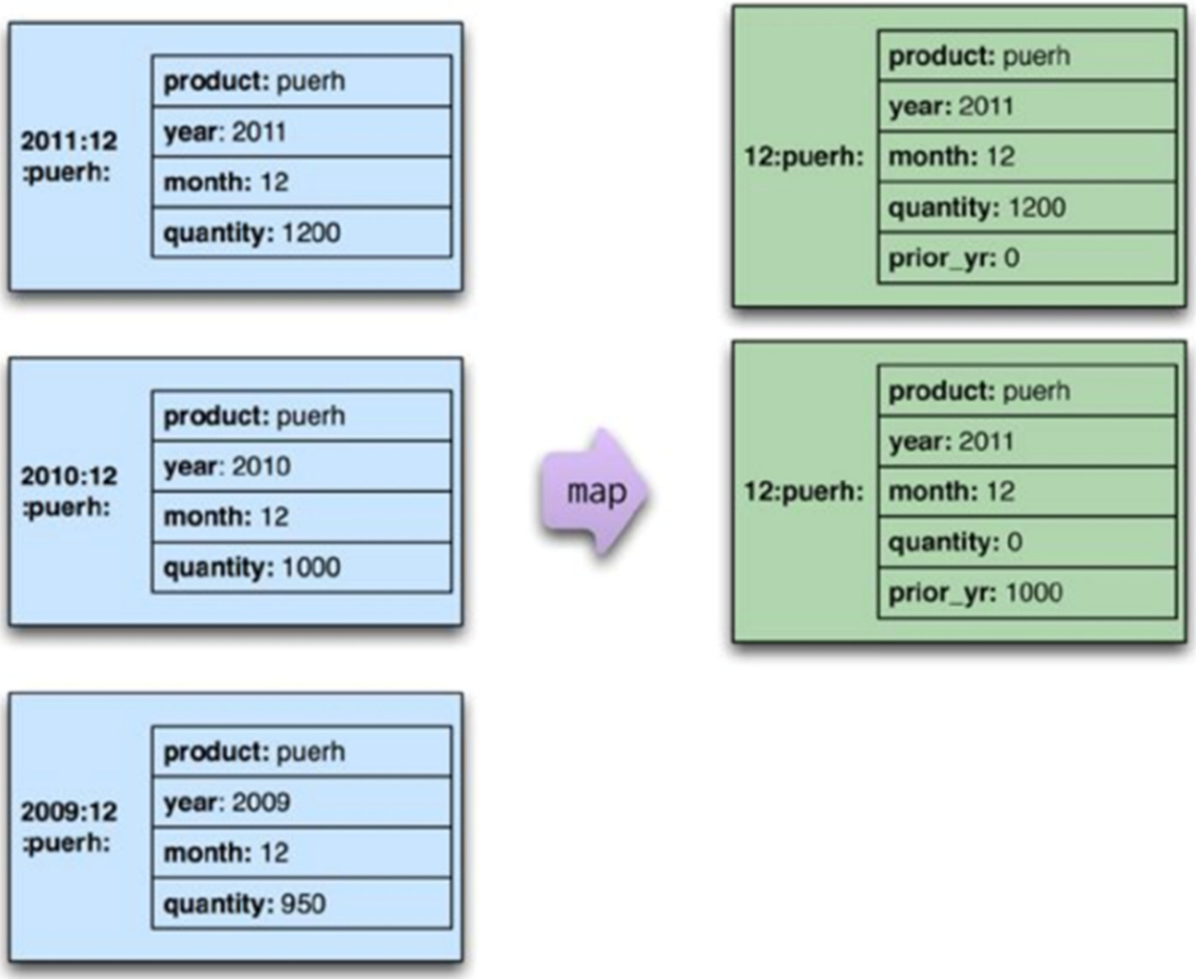
- As map -reduce calculations get more complex, it's useful to break them down into stages.
- Consider an example where we want to compare the sales of products for each month in 2011 to the prior year.
- To do this, we'll break the calculations down into two stages.
- The first stage will produce records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

A two stage Map-Reduce example



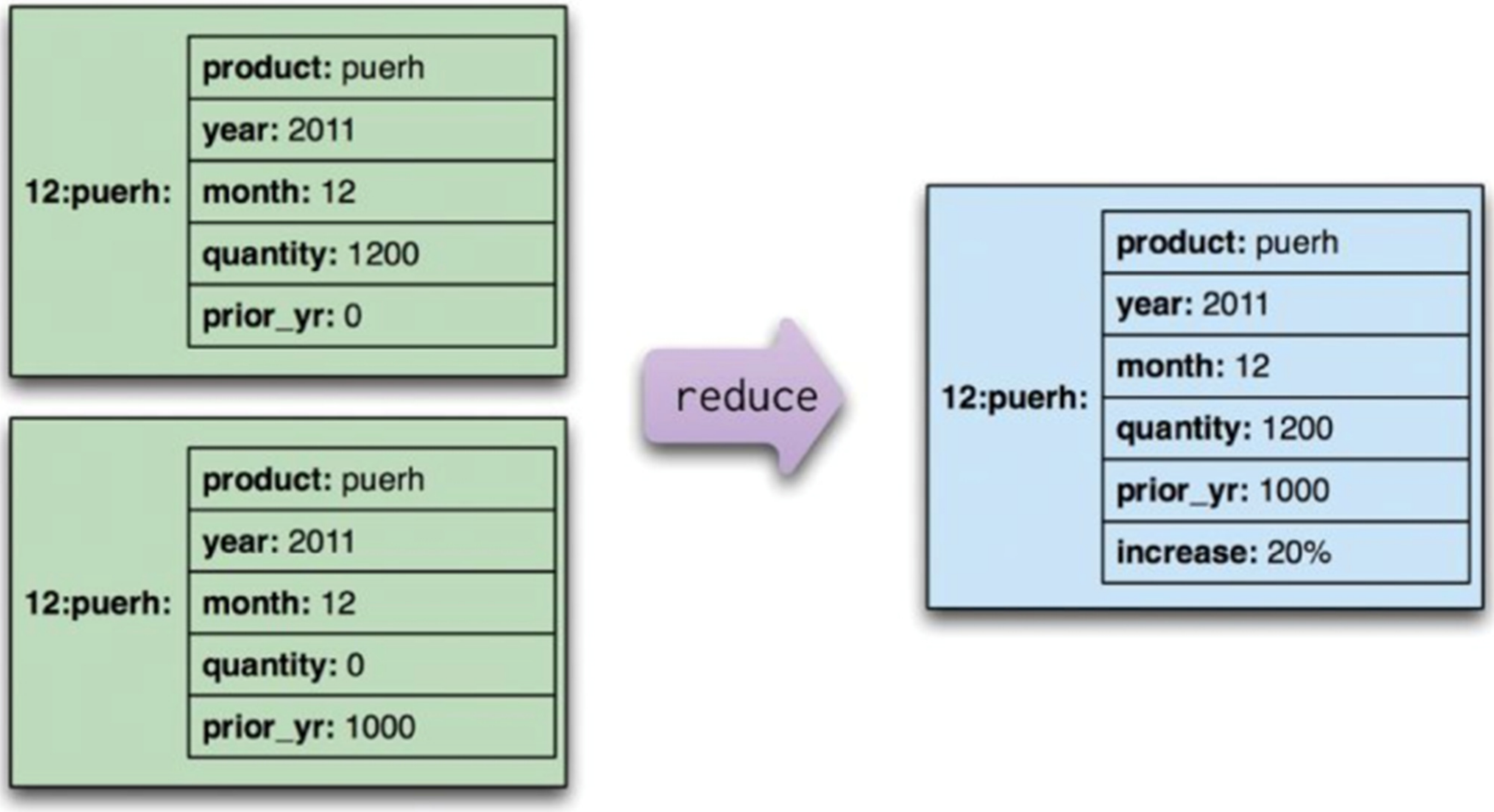
First stage: Creating records for monthly sales of a product

A two stage Map-Reduce example



Second stage (map) : mappers take as input the output of the first stage, and "prepare" the input for reducers

A two stage Map-Reduce example



Second stage (reduce) : The reduction step is a merge of incomplete records.

Map-reduce – further aspects

- Map-reduce is a pattern that can be implemented in any programming language.
- But, it is a good fit for languages specifically designed for map-reduce computations.
- Apache Pig , an offshoot of the Hadoop project , is a language specifically built to make it easy to write map-reduce programs
- In a similar vein, if you want to specify map-reduce programs using an SQL-like syntax, there is Hive, another Hadoop offshoot (it takes your SQL-like queries and turns them into MapReduce jobs)
- Another interesting tool in the Hadoop ecosystem is Sqoop, which integrates with any JDBC-compatible database (import into HDFS from a RDBMS, and export to a RDBMS)