Module-3

Chapter 1 - Map Reduce

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

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

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

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

1.1 Basic Map-Reduce

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

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

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

pairs. In this case, the input would be an order. The output would be key-value pairs corresponding to the line items. Each one would have the product ID as the key and an embedded map with the quantity and price as the values (see Figure 1.1).

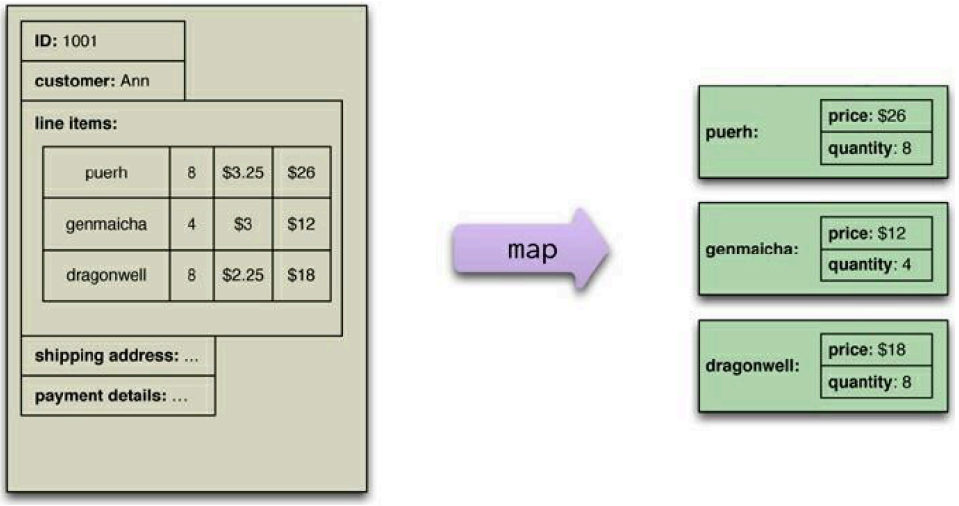

Figure 1.1. A map function reads records from the database and emits key-value pairs.

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

A map operation only operates on a single record; the reduce function takes multiple map outputs with the same key and combines their values.

So, a map function might yield 1000 line items from orders for "Database Refactoring"; the reduce function would reduce down to one, with the totals for the quantity and revenue. While the map function is limited to working only on data from a single aggregate, the reduce function can use all values emitted for a single key (see Figure 1.2).

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

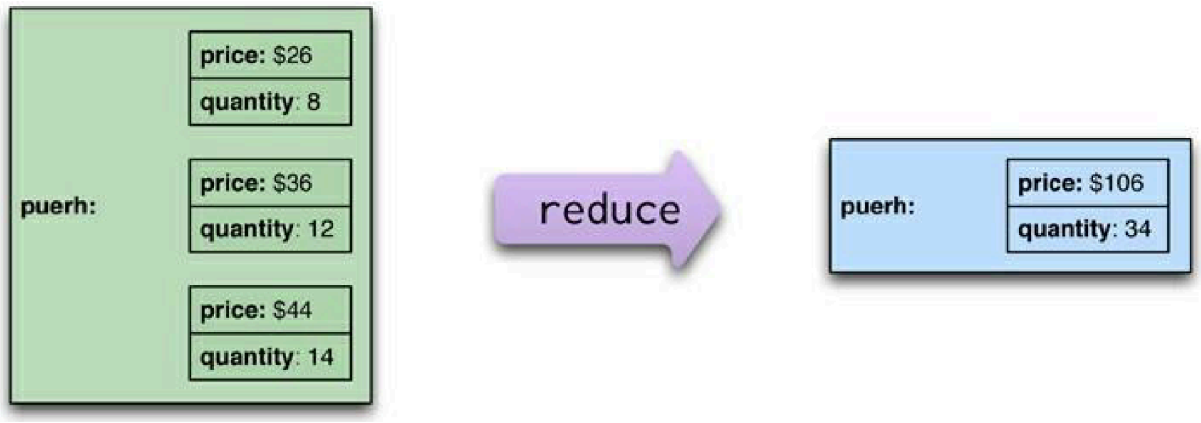

Figure 1.2. A reduce function takes several key-value pairs with the same key and aggregates them into one.

1.2 Partitioning and Combining

In the simplest form, we think of a map-reduce job as having a single reduce function. The outputs from all the map tasks running on the various nodes are concatenated together and sent into the reduce. While this will work, there are things we can do to increase the parallelism and to reduce the data transfer(see figure 1.3)

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

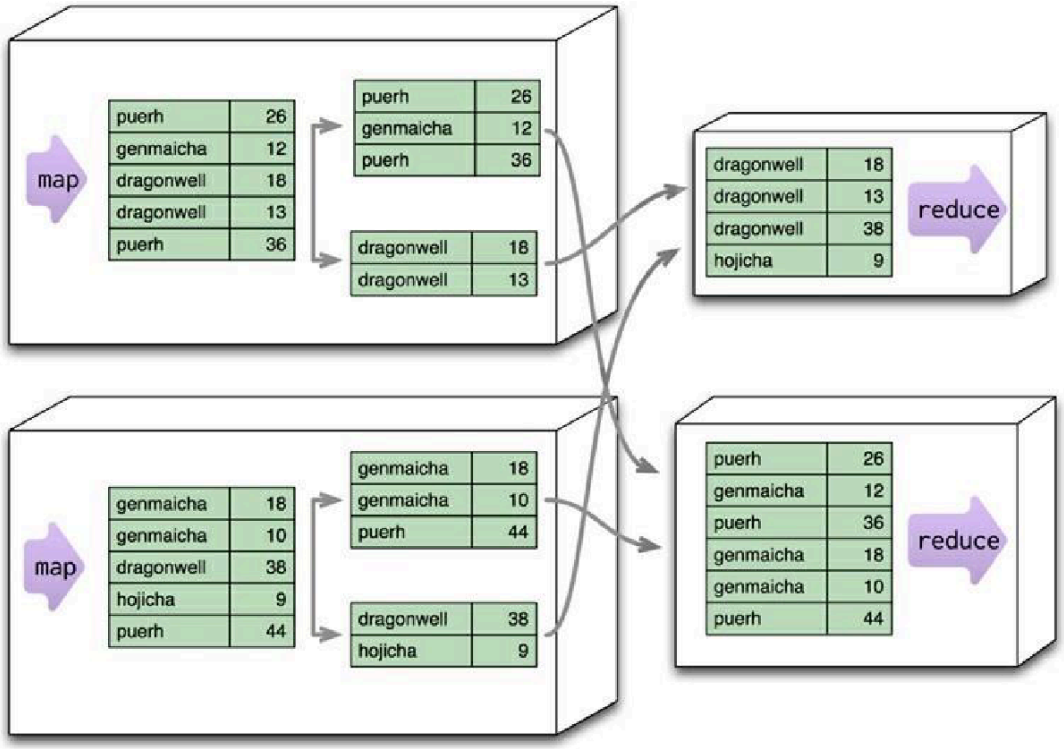

Figure 1.3. Partitioning allows reduce functions to run in parallel on different keys.

The next problem we can deal with is the amount of data being moved from node to node between the map and reduce stages. Much of this data is repetitive, consisting of multiple key-value pairs for the same key. A combiner function cuts this data down by combining all the data for the same key into a single value (see Figure 1.4). A combiner function is, in essence, a reducer function—indeed, in many cases the same function can be used for combining as the final reduction. The reduce function needs a special shape for this to work: Its output must match its input. We call such a function a combinable reducer.

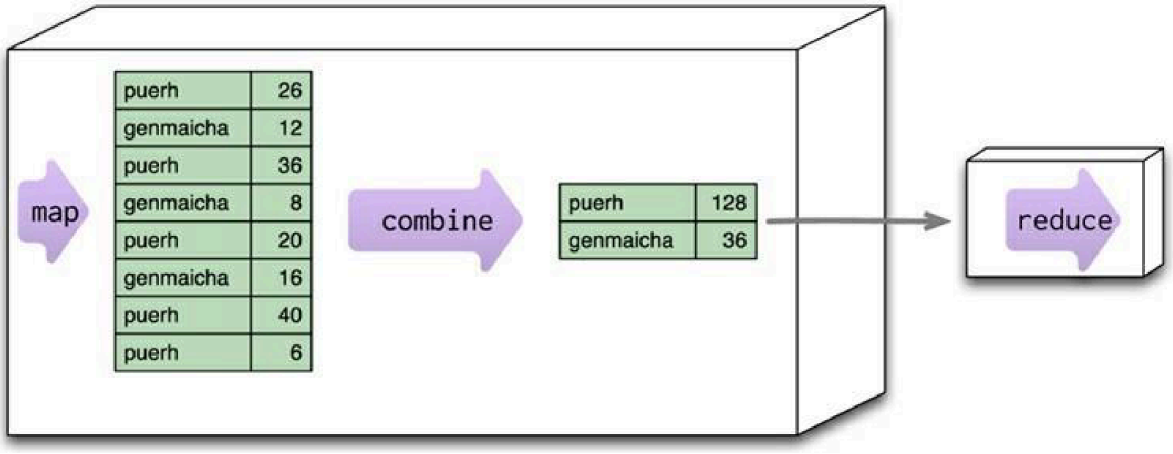

Figure 1.4. Combining reduces data before sending it across the network.

Not all reduce functions are combinable. Consider a function that counts the number of unique customers

for a particular product. The map function for such an operation would need to emit the

product and the customer. The reducer can then combine them and count how many times each customer appears for a particular product, emitting the product and the count (see Figure 1.5). But this reducer's output is different from its input, so it can't be used as a combiner. You can still run a combining function here: one that just eliminates duplicate product-customer pairs, but it will be different from the final reducer.

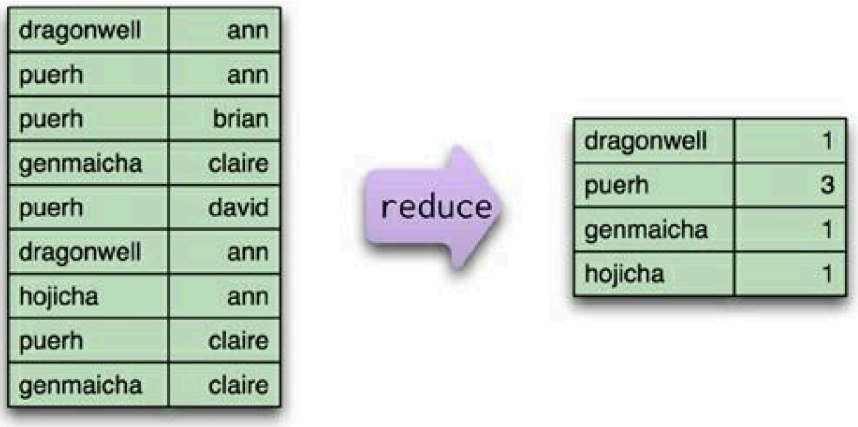

Figure 1.5. This reduce function, which counts how many unique customers order a particular tea, is not combinable.

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

1.3 Composing Map-Reduce Calculations

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

One simple limitation is that you have to structure your calculations around operations that fit in well with the notion of a reduce operation. A good example of this is calculating averages. Let's consider the kind of orders we've been looking at so far; suppose we want to know the average ordered quantity of each

product. An important property of averages is that they are not comparable.

that is, if I take two groups of orders, I can't combine their averages alone. Instead, I need to take total amount and the count of orders from each group, combine those, and then calculate the average from the combined sum and count (see Figure 1.6).

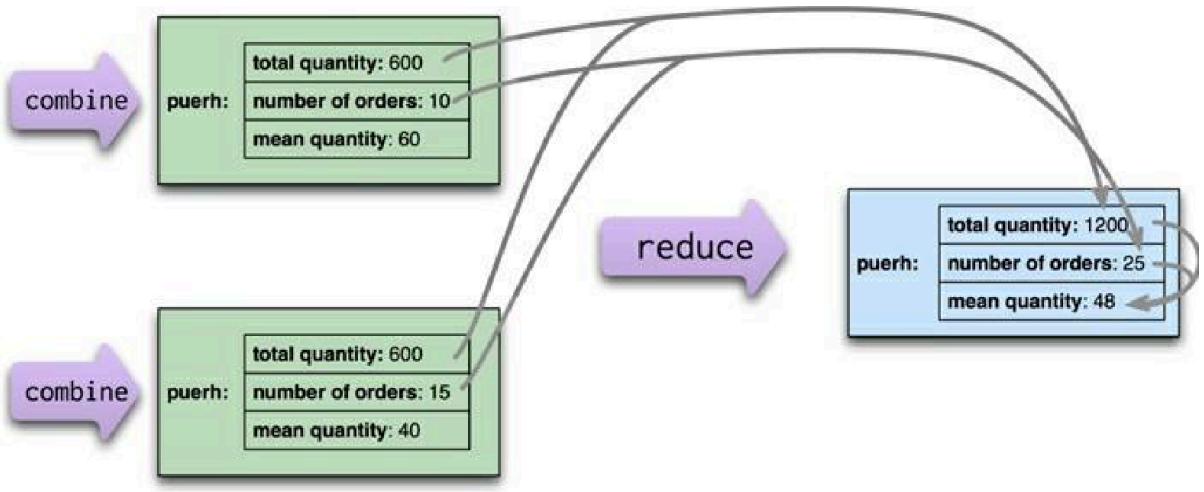

Figure 1.6. When calculating averages, the sum and count can be combined in the reduce calculation, but the average must be calculated from the combined sum and count.

This notion of looking for calculations that reduce neatly also affects how we do counts. To make a count, the mapping function will emit count fields with a value of 1, which can be summed to get a total count (see Figure 1.7).

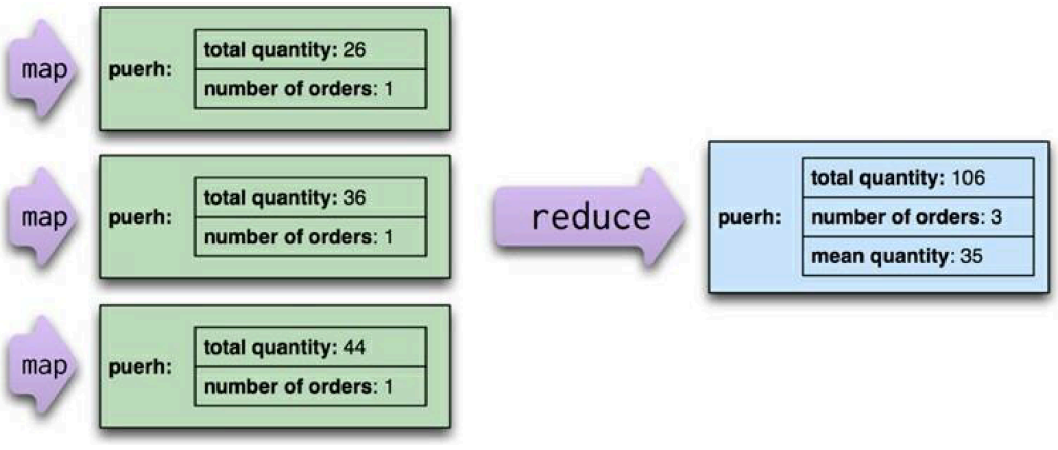

Figure 1.7. When making a count, each map emits 1, which can be summed to get a total.

1.4 A Two Stage Map-Reduce Example

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

Consider an example where we want to compare the sales of products for each month in 2011 to the prior year. To do this, we'll break the calculations down into two stages. The first stage will produce records showing the aggregate figures for a single product in a single month of the year. The second stage then uses these as inputs and produces the result for a single product by comparing one month's results with the same month in the prior year (see Figure 1.8).

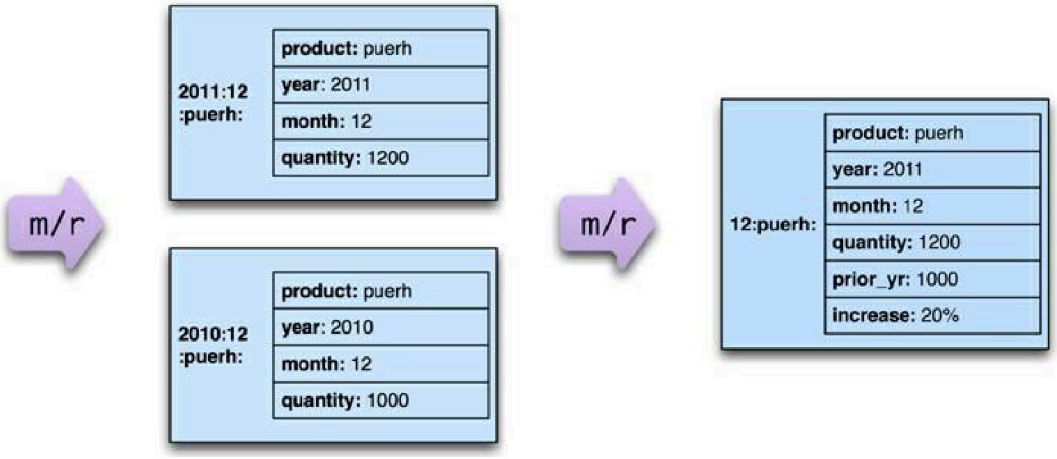

Figure 1.8. A calculation broken down into two map-reduce steps, which will be expanded in the next three figures

A first stage (Figure 1.9) would read the original order records and output a series of key-value pairs for the sales of each product per month.

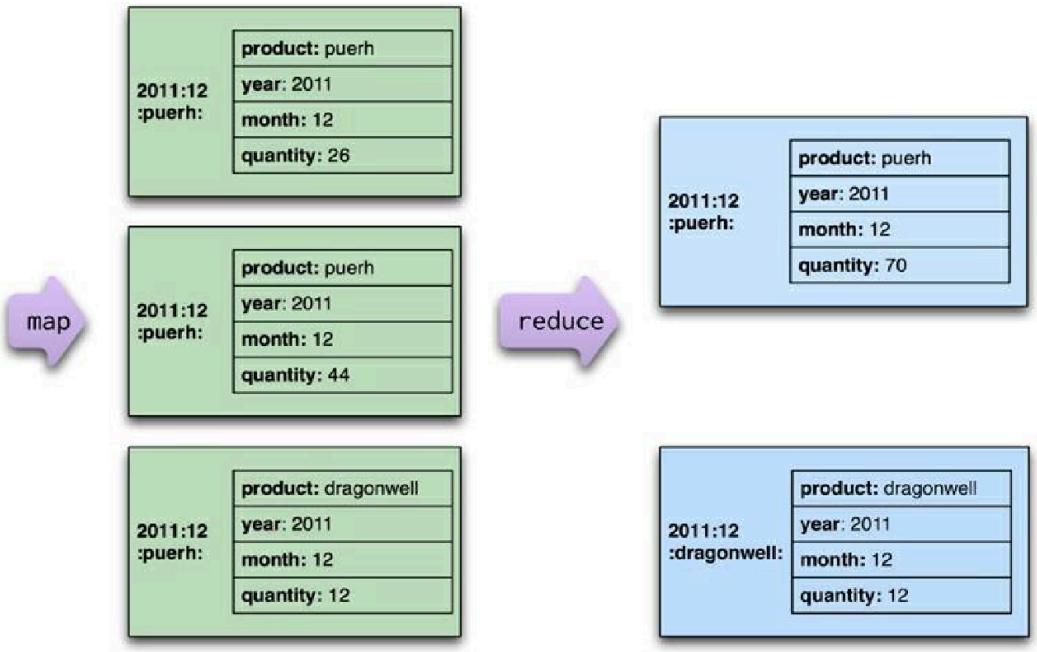

Figure 1.9. Creating records for monthly sales of a product

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

The second-stage mappers (Figure 1.10) process this output depending on the year. A 2011 record populates the current year quantity while a 2010 record populates a prior year quantity. Records for earlier years (such as 2009) don't result in any mapping output being emitted.

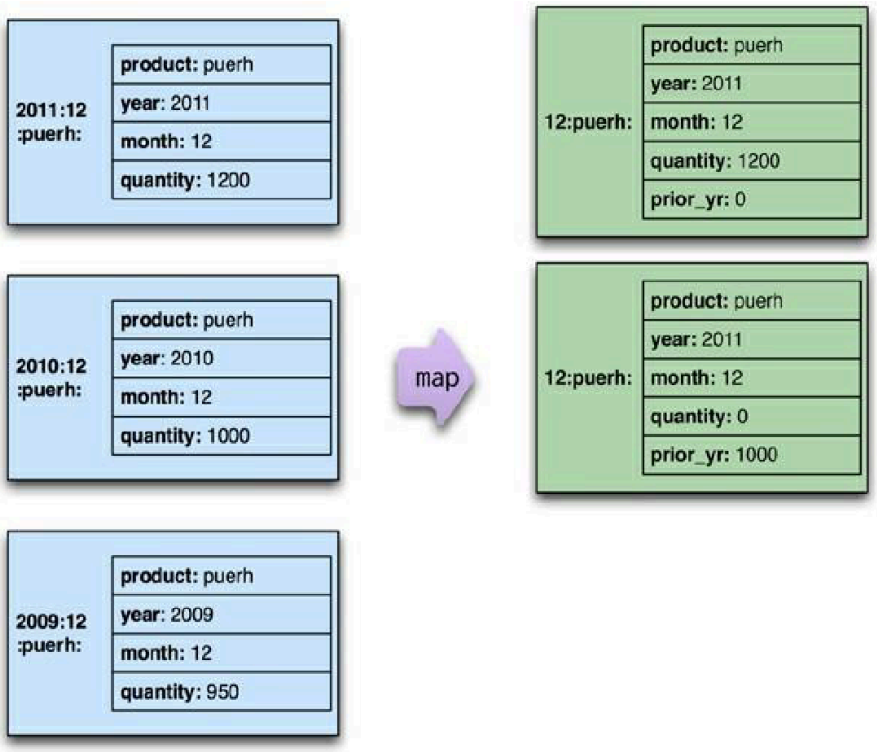

Figure 1.10. The second stage mapper creates base records for year-on-year comparisons.

The reduce in this case (Figure 1.11) is a merge of records, where combining the values by summing allows two different year outputs to be reduced to a single value (with a calculation based on the reduced values thrown in for good measure).

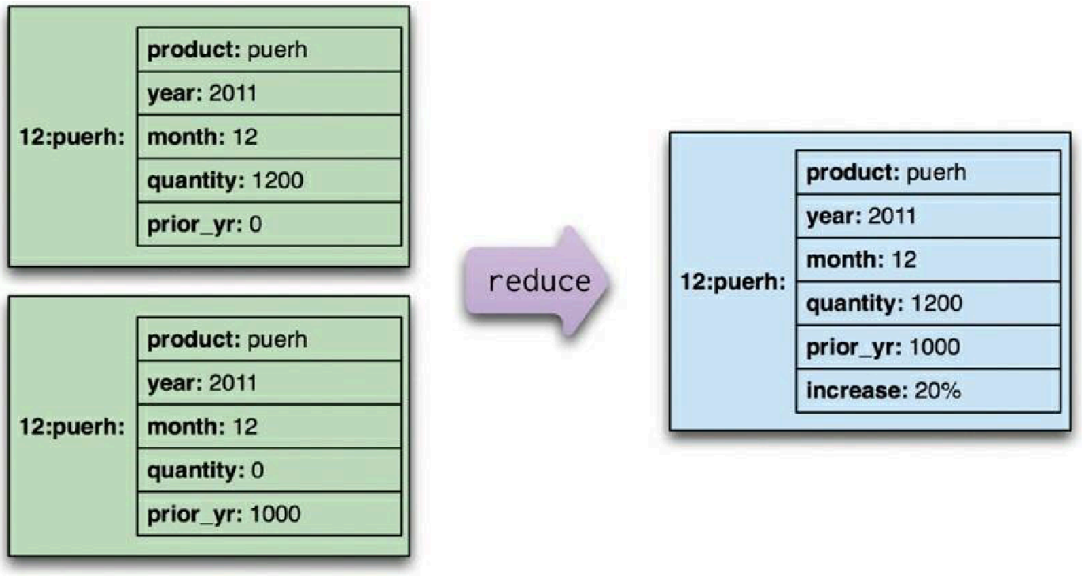

Figure 1.11. The reduction step is a merge of incomplete records.

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

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

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

The map-reduce pattern is important to know about even outside of the context of NoSQL databases. Google's original map-reduce system operated on files stored on a distributed file system an approach that's used by the open-source Hadoop project. While it takes some thought to get used to the constraints

of structuring computations in map-reduce steps, the result is a calculation that is inherently well-suited to running on a cluster. When dealing with high volumes of data, you need to take a cluster-oriented approach. Aggregate-oriented databases fit well with this style of calculation. We think that in the next few years many more organizations will be processing the volumes of data that demand a cluster-oriented solution—and the map-reduce pattern will see more and more use.

1.5 Incremental Map-Reduce

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

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

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

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

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

Chapter 2 - Key-Value Databases

A key-value store is a simple hash table, primarily used when all access to the database is via primary key. Think of a table in a traditional RDBMS with two columns, such as ID and NAME, the ID column being the key and NAME column storing the value. In an RDBMS, the NAME column is restricted to storing

data of type String. The application can provide an ID and VALUE and persist the pair; if the ID already exists the current value is overwritten, otherwise a new entry is created. Let's look at how terminology compares in Oracle and Riak.

Oracle	Riak
database instance	Riak cluster
table	bucket
row	key-value
rowid	key

2.1 What Is a Key-Value Store

Key-value stores are the simplest NoSQL data stores to use from an API perspective. The client can either get the value for the key, put a value for a key, or delete a key from the data store. The value is a blob that the data store just stores, without caring or knowing what's inside; it's the responsibility of the application to understand what was stored. Since key-value stores always use primary-key access, they generally have great performance and can be easily scaled.

Some of the popular key-value databases are Riak [Riak], Redis (often referred to as Data Structure server) [Redis], Memcached DB and its flavors [Memcached], Berkeley DB [Berkeley DB], HamsterDB (especially suited for embedded use) [HamsterDB], Amazon DynamoDB [Amazon's Dynamo] (not open-source), and Project Voldemort [Project Voldemort] (an open-source implementation of Amazon DynamoDB).

In some key-value stores, such as Redis, the aggregate being stored does not have to be a domain object—it could be any data structure. Redis supports storing lists, sets, hashes and can do range, diff, union, and intersection operations. These features allow Redis to be used in more different ways than a standard key-value store.

There are many more key-value databases and many new ones are being worked on at this time. For the sake of keeping discussions in this book easier we will focus mostly on Riak. Riak lets us store keys into buckets, which are just a way to segment the keys—think of buckets as flat namespaces for the keys.

If we wanted to store user session data, shopping cart information, and user preferences in Riak, we could just store all of them in the same bucket with a single key and single value for all of these objects. In this scenario, we would have a single object that stores all the data and is put into a single bucket (Figure 2.1).

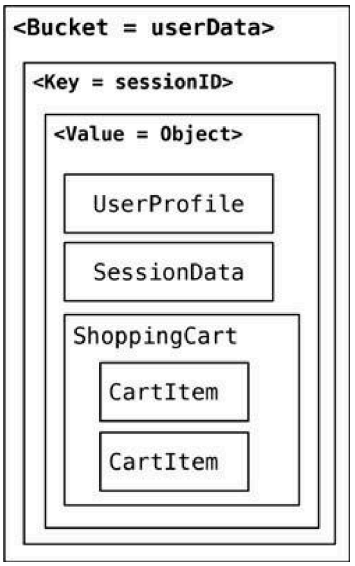

Figure 2.1. Storing all the data in a single bucket

The downside of storing all the different objects (aggregates) in the single bucket would be that one bucket would store different types of aggregates, increasing the chance of key conflicts. An alternate approach would be to append the name of the object to the key, such as 288790b8a421_userProfile, so that we can get to individual objects as they are needed.

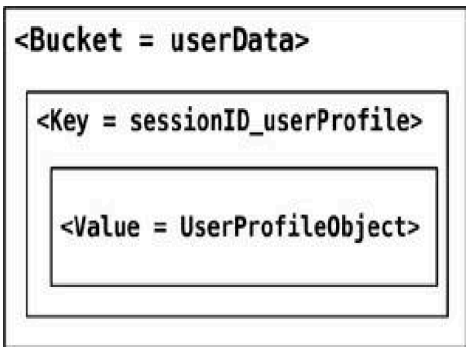

Figure 2.2. Change the key design to segment the data in a single bucket.

We could also create buckets which store specific data. In Riak, they are known as domain buckets allowing the serialization and deserialization to be handled by the client driver.

Bucket bucket = client.fetchBucket(bucketName).execute();

DomainBucket<UserProfile> profileBucket = DomainBucket.builder(bucket, UserProfile.class).build();

Using domain buckets or different buckets for different objects (such as UserProfile and ShoppingCart) segments the data across different buckets allowing you to read only the object you need without having to change key design.

Key-value stores such as Red is also support storing random data structures, which can be sets, hashes, strings, and so on. This feature can be used to store lists of things, like states or address Types, or an array

of user's visits.

2.1.1 Key-Value Store Features

While using any NoSQL data stores, there is an inevitable need to understand how the features compare to the standard RDBMS data stores that we are so used to. The primary reason is to understand what features are missing and how does the application architecture need to change to better use the features of a key-value data store. Some of the features we will discuss for all the NoSQL data stores are consistency, transactions, query features, structure of the data, and scaling.

Consistency

Consistency is applicable only for operations on a single key, since these operations are either a get, put, or delete on a single key. Optimistic writes can be performed, but are very expensive to implement, because a change in value cannot be determined by the data store.

In distributed key-value store implementations like Riak, the eventually consistent (p. 50) model of consistency is implemented. Since the value may have already been replicated to other nodes, Riak has two ways of resolving update conflicts: either the newest write wins and older writes loose, or both (all) values are returned allowing the client to resolve the conflict.

In Riak, these options can be set up during the bucket creation. Buckets are just a way to namespace keys so that key collisions can be reduced—for example, all customer keys may reside in the customer bucket. When creating a bucket, default values for consistency can be provided, for example that a write is considered good only when the data is consistent across all the nodes where the data is stored.

```
Bucket bucket = connection

.createBucket(bucketName)

.withRetrier(attempts(3))

.allowSiblings(siblingsAllowed)

.nVal(numberOfReplicasOfTheData)

.w(numberOfNodesToRespondToWrite)

.r(numberOfNodesToRespondToRead)

.execute();
```

If we need data in every node to be consistent, we can increase the numberOfNodesToRespondToWrite set by w to be the same as nVal. Of course doing that will decrease the write performance of the cluster. To improve on write or read conflicts, we can change the allowSiblings flag during bucket creation: If it is

set to false, we let the last write to win and not create siblings.

Transactions

Different products of the key-value store kind have different specifications of transactions. Generally speaking, there are no guarantees on the writes. Many data stores do implement transactions in different ways. Riak uses the concept of quorum (“Quorums,” p. 57) implemented by using the W value —replication factor—during the write API call.

Assume we have a Riak cluster with a replication factor of 5 and we supply the W value of 3. When writing, the write is reported as successful only when it is written and reported as a success on at least three of the nodes. This allows Riak to have write tolerance; in our example, with N equal to 5 and with a W value of 3, the cluster can tolerate N - W = 2 nodes being down for write operations, though we would still have lost some data on those nodes for read.

Query Features

All key-value stores can query by the key—and that’s about it. If you have requirements to query by using some attribute of the value column, it’s not possible to use the database: Your application needs to read the value to figure out if the attribute meets the conditions.

Query by key also has an interesting side effect. What if we don’t know the key, especially during ad-hoc querying during debugging? Most of the data stores will not give you a list of all the primary keys; even if they did, retrieving lists of keys and then querying for the value would be very cumbersome. Some key-value databases get around this by providing the ability to search inside the value, such as Riak Search that allows you to query the data just like you would query it using Lucene indexes.

While using key-value stores, lots of thought has to be given to the design of the key. Can the key be generated using some algorithm? Can the key be provided by the user (user ID, email, etc.)? Or derived from timestamps or other data that can be derived outside of the database?

These query characteristics make key-value stores likely candidates for storing session data (with the session ID as the key), shopping cart data, user profiles, and so on. The expiry_secs property can be used to expire keys after a certain time interval, especially for session/shopping cart objects.

Bucket bucket = getBucket(bucketName);

IRiakObject riakObject = bucket.store(key, value).execute();

When writing to the Riak bucket using the store API, the object is stored for the key provided.

Similarly, we can get the value stored for the key using the fetch API.

```
Bucket bucket = getBucket(bucketName);

IRiakObject riakObject = bucket.fetch(key).execute(); byte[] bytes = riakObject.getValue();

String value = new String(bytes);
```

Riak provides an HTTP-based interface, so that all operations can be performed from the web browser or on the command line using curl. Let's save this data to Riak:

```
{

"lastVisit":1324669989288, "user":{

"customerId":"91cfdf5bcb7c", "name":"buyer",

"countryCode":"US", "tzOffset":0

}

}
```

Use the curl command to POST the data, storing the data in the session bucket with the key of

a7e618d9db25 (we have to provide this key):

```
curl -v -X POST -d '

{ "lastVisit":1324669989288,

"user":{"customerId":"91cfdf5bcb7c", "name":"buyer",

"countryCode":"US", "tzOffset":0}

}'

-H "Content-Type: application/json" http://localhost:8098/buckets/session/keys/a7e618d9db25
```

The data for the key a7e618d9db25 can be fetched by using the curl command: `curl -i http://localhost:8098/buckets/session/keys/a7e618d9db25`

Structure of Data

Key-value databases don't care what is stored in the value part of the key-value pair. The value can be a blob, text, JSON, XML, and so on. In Riak, we can use the Content-Type in the POST request to specify the data type.

Scaling

Many key-value stores scale by using sharding ("Sharding," p. 38). With sharding, the value of the key determines on which node the key is stored. Let's assume we are sharding by the first character of the

key; if the key is f4b19d79587d, which starts with an f, it will be sent to different node than the key ad9c7a396542. This kind of sharding setup can increase performance as more nodes are added to the cluster.

Sharding also introduces some problems. If the node used to store f goes down, the data stored on that node becomes unavailable, nor can new data be written with keys that start with f.

Data stores such as Riak allow you to control the aspects of the CAP Theorem ("The CAP Theorem," p. 53): N (number of nodes to store the key-value replicas), R (number of nodes that have to have the data being fetched before the read is considered successful), and W (the number of nodes the write has to be written to before it is considered successful).

Let's assume we have a 5-node Riak cluster. Setting N to 3 means that all data is replicated to at least three nodes, setting R to 2 means any two nodes must reply to a GET request for it to be considered successful and setting W to 2 ensures that the PUT request is written to two nodes before the write is considered successful.

These settings allow us to fine-tune node failures for read or write operations. Based on our need, we can change these values for better read availability or write availability. Generally speaking choose a W value to match your consistency needs; these values can be set as defaults during bucket creation.

Suitable Use Cases

Let's discuss some of the problems where key-value stores are a good fit.

Storing Session Information

Generally, every web session is unique and is assigned a unique sessionid value. Applications that store the sessionid on disk or in an RDBMS will greatly benefit from moving to a key-value store, since everything about the session can be stored by a single PUT request or retrieved using GET. This single-request operation makes it very fast, as everything about the session is stored in a single object. Solutions such as Memcached are used by many web applications, and Riak can be used when availability is important.

User Profiles, Preferences

Almost every user has a unique userId, username, or some other attribute, as well as preferences such as language, color, timezone, which products the user has access to, and so on. This can all be put into an object, so getting preferences of a user takes a single GET operation. Similarly, product profiles can be stored.

Shopping Cart Data

E-commerce websites have shopping carts tied to the user. As we want the shopping carts to be available all the time, across browsers, machines, and sessions, all the shopping information can be put into the value where the key is the userid. A Riak cluster would be best suited for these kinds of applications.

When Not to Use

There are problem spaces where key-value stores are not the best solution.

Relationships among Data

If you need to have relationships between different sets of data, or correlate the data between different sets of keys, key-value stores are not the best solution to use, even though some key-value stores provide link-walking features.

Multioperation Transactions

If you're saving multiple keys and there is a failure to save any one of them, and you want to revert or roll back the rest of the operations, key-value stores are not the best solution to be used.

Query by Data

If you need to search the keys based on something found in the value part of the key-value pairs, then key-value stores are not going to perform well for you. There is no way to inspect the value on the database side, with the exception of some products like Riak Search or indexing engines like Lucene [Lucene] or Solr [Solr].

Operations by Sets

Since operations are limited to one key at a time, there is no way to operate upon multiple keys at the same time. If you need to operate upon multiple keys, you have to handle this from the client side.