Chapter 7. Map-Reduce

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

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

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

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

7.1. Basic Map-Reduce

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

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

This is exactly the kind of situation that calls for map-reduce. The first stage in a map-reduce job is the map. A map is a function whose input is a single aggregate and whose output is a bunch of key-value pairs. In this case, the input would be an order. The output would be key-value pairs corresponding to the line items. Each one would have the product ID as the key and an embedded map with the quantity and price as the values (see Figure 7.1).

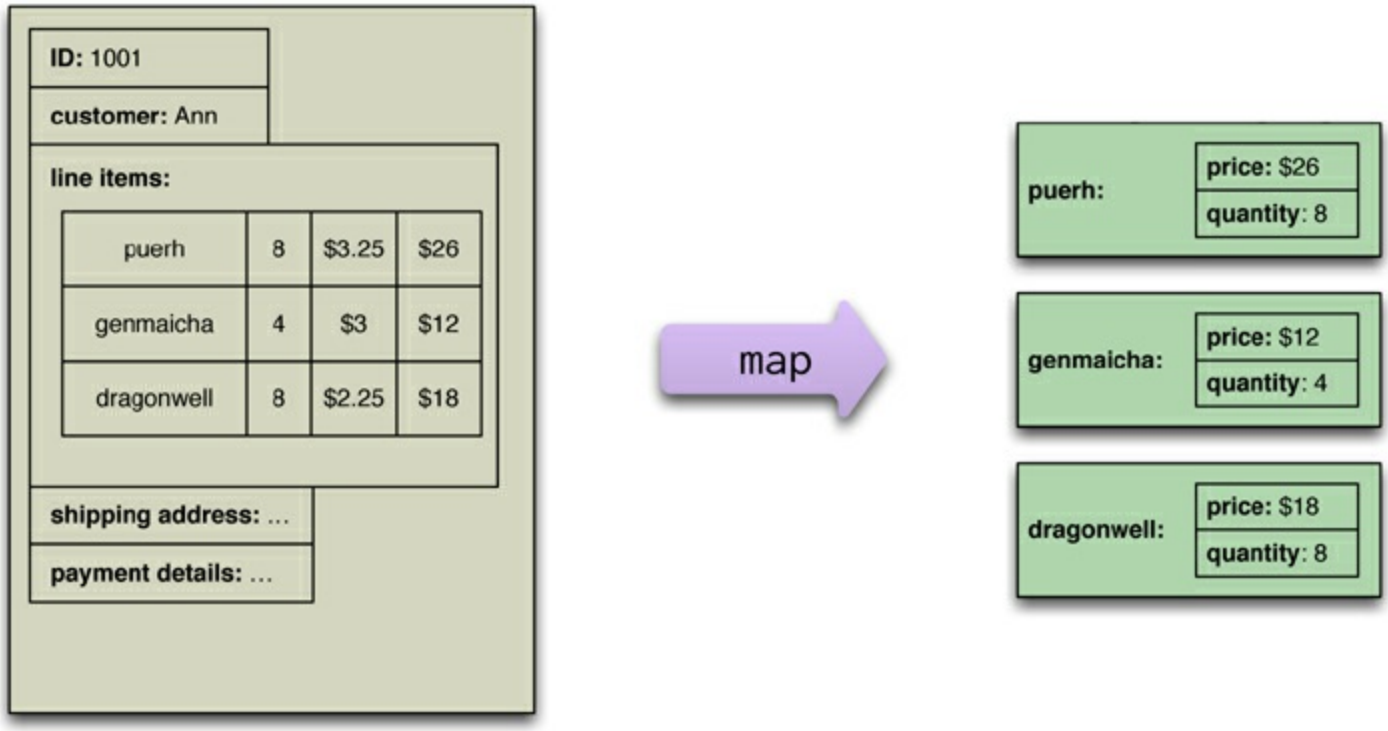

Figure 7.1. A map function reads records from the database and emits key-value pairs.

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

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

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

The map-reduce framework arranges for map tasks to be run on the correct nodes to process all the documents and for data to be moved to the reduce function. To make it easier to write the reduce function, the framework collects all the values for a single pair and calls the reduce function once

with the key and the collection of all the values for that key. So to run a map-reduce job, you just need to write these two functions.

7.2. Partitioning and Combining

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

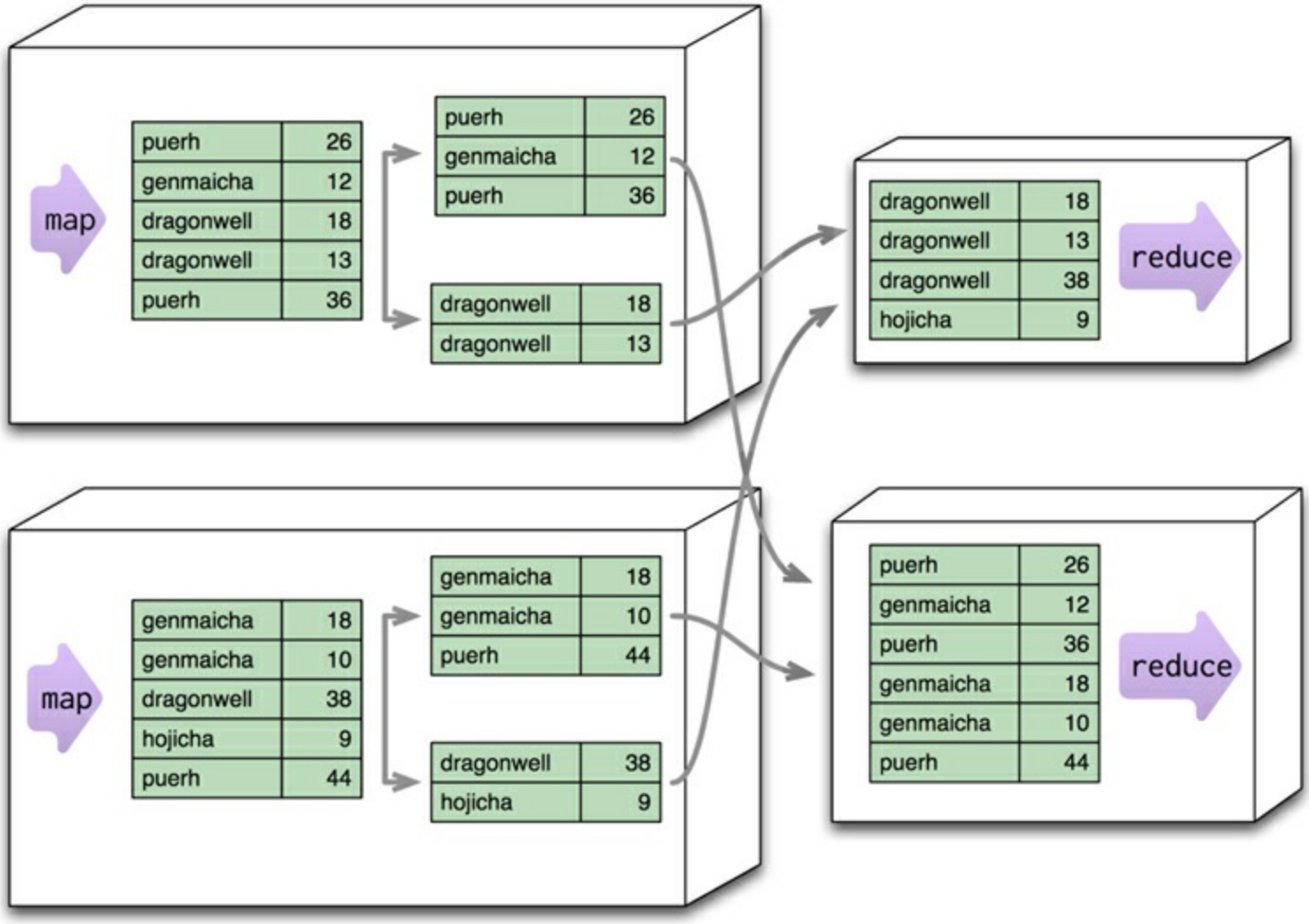

Figure 7.3. Partitioning allows reduce functions to run in parallel on different keys.

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

The next problem we can deal with is the amount of data being moved from node to node between the map and reduce stages. Much of this data is repetitive, consisting of multiple key-value pairs for the same key. A combiner function cuts this data down by combining all the data for the same key into

a single value (see Figure 7.4). A combiner function is, in essence, a reducer function—indeed, in many cases the same function can be used for combining as the final reduction. The reduce function needs a special shape for this to work: Its output must match its input. We call such a function a **combinable reducer**.

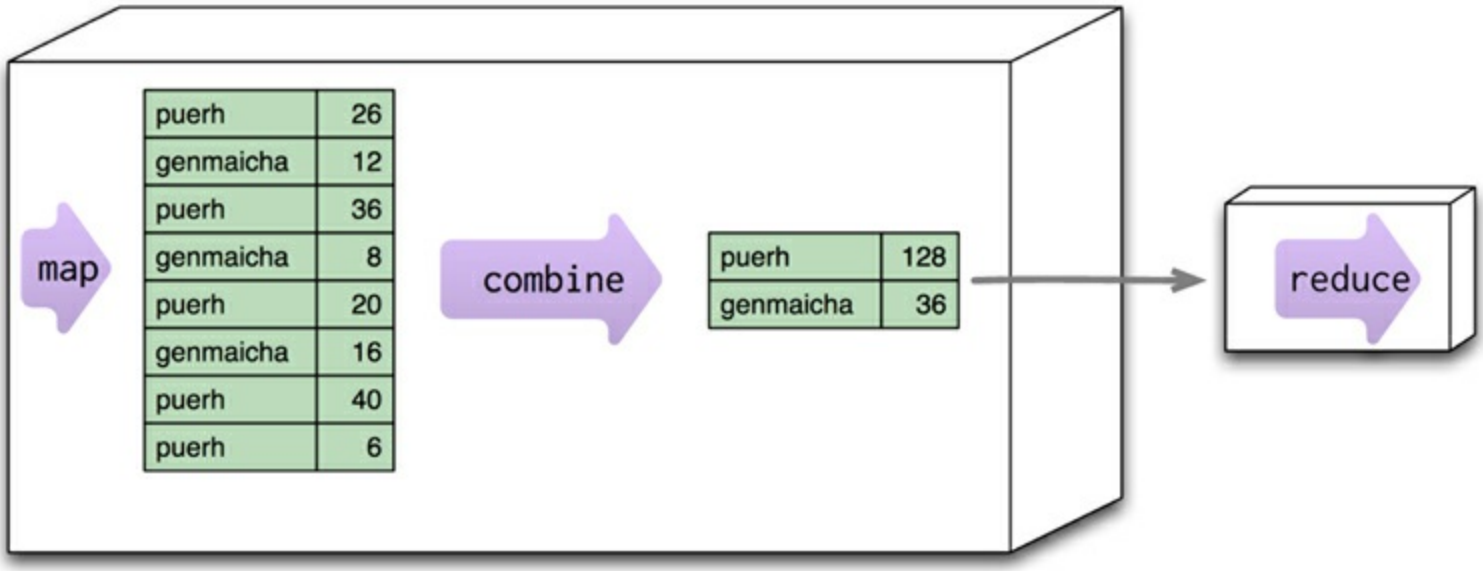

Figure 7.4. Combining reduces data before sending it across the network.

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

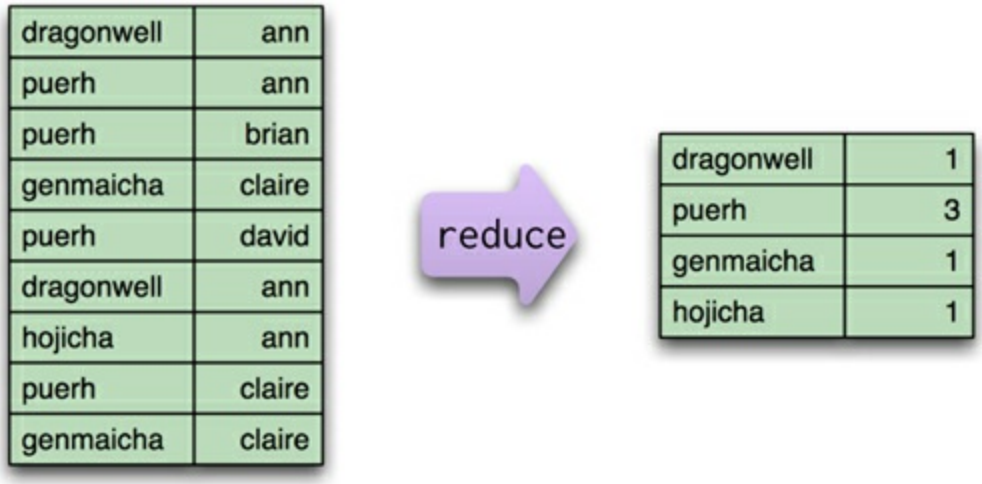

Figure 7.5. This reduce function, which counts how many unique customers order a particular tea, is not combinable.

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

these frameworks, you'll need to separate the processing into pipelined map-reduce steps.

7.3. Composing Map-Reduce Calculations

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

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

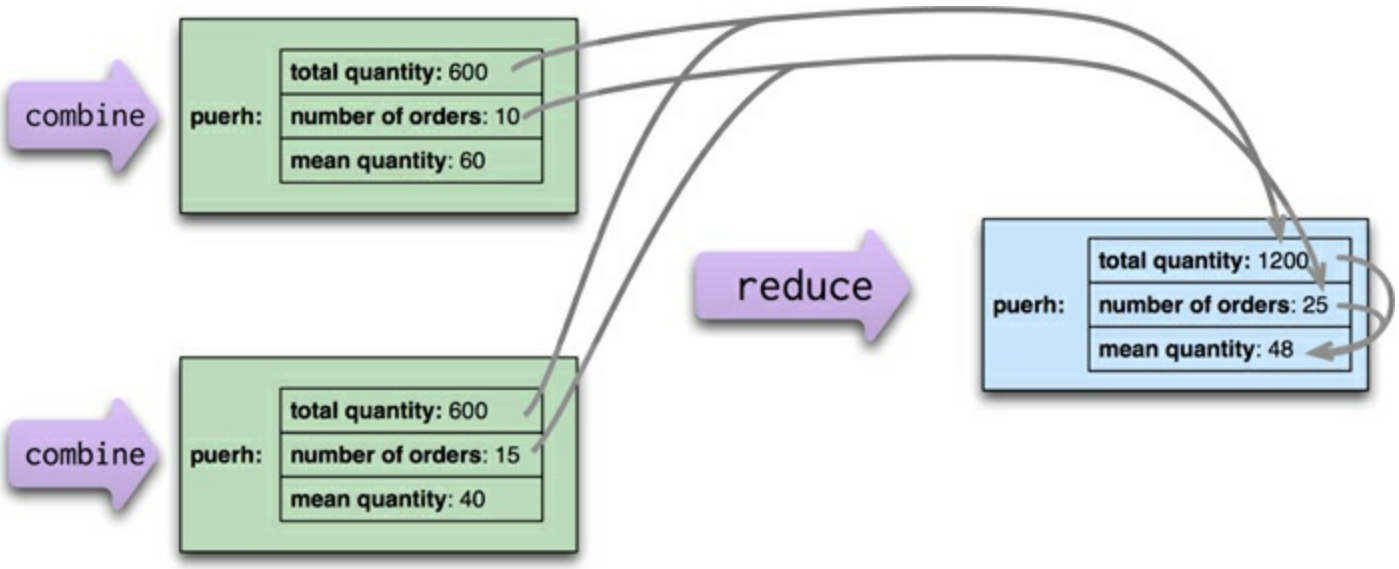

Figure 7.6. When calculating averages, the sum and count can be combined in the reduce calculation, but the average must be calculated from the combined sum and count.

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

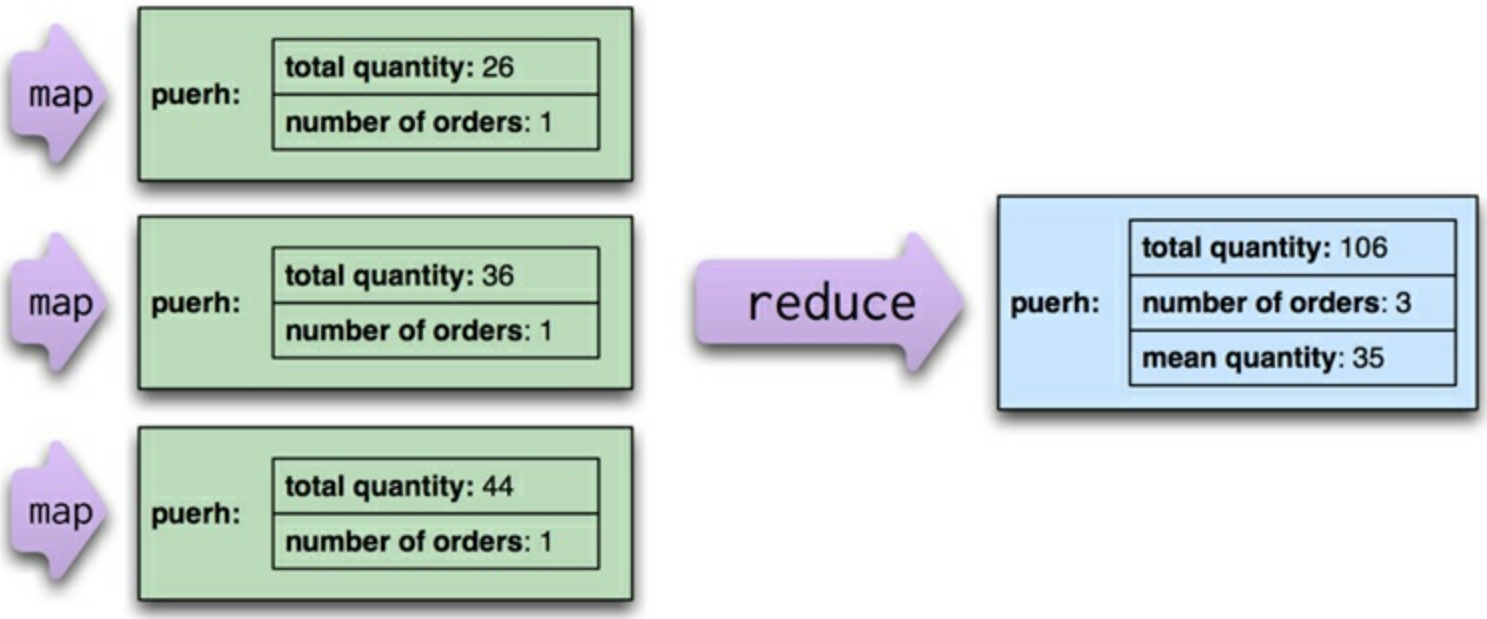

Figure 7.7. When making a count, each map emits 1, which can be summed to get a total.

7.3.1. A Two Stage Map-Reduce Example

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

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

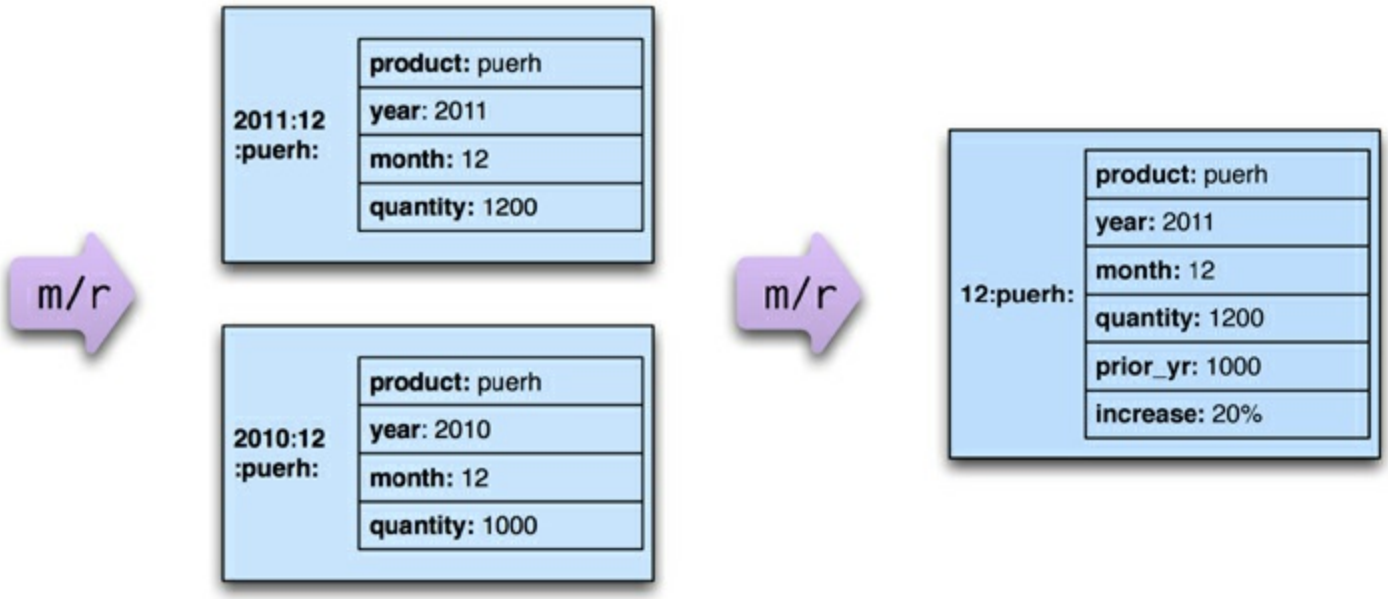

Figure 7.8. A calculation broken down into two map-reduce steps, which will be expanded in the next three figures

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

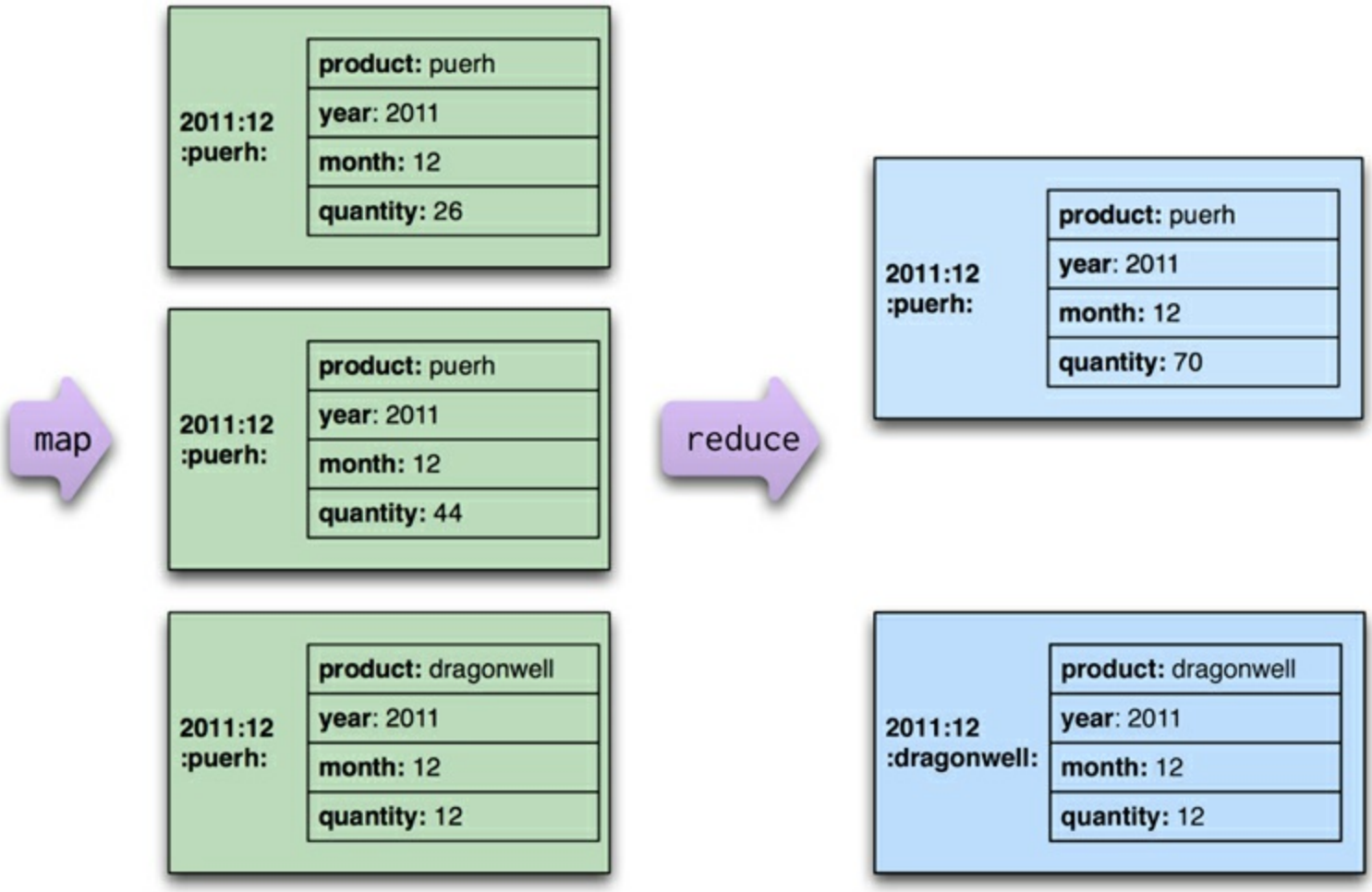

Figure 7.9. Creating records for monthly sales of a product

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

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

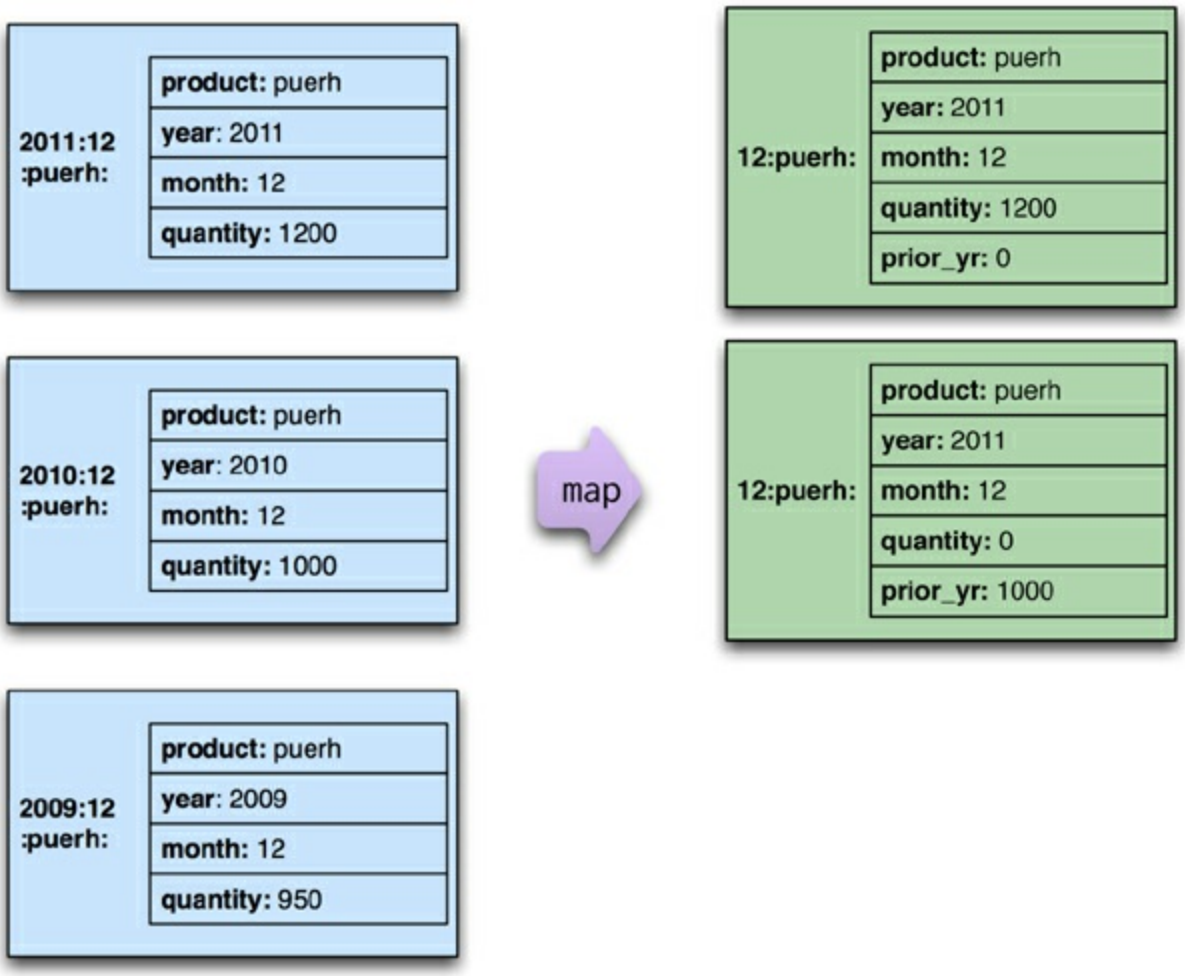

Figure 7.10. The second stage mapper creates base records for year-on-year comparisons.

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

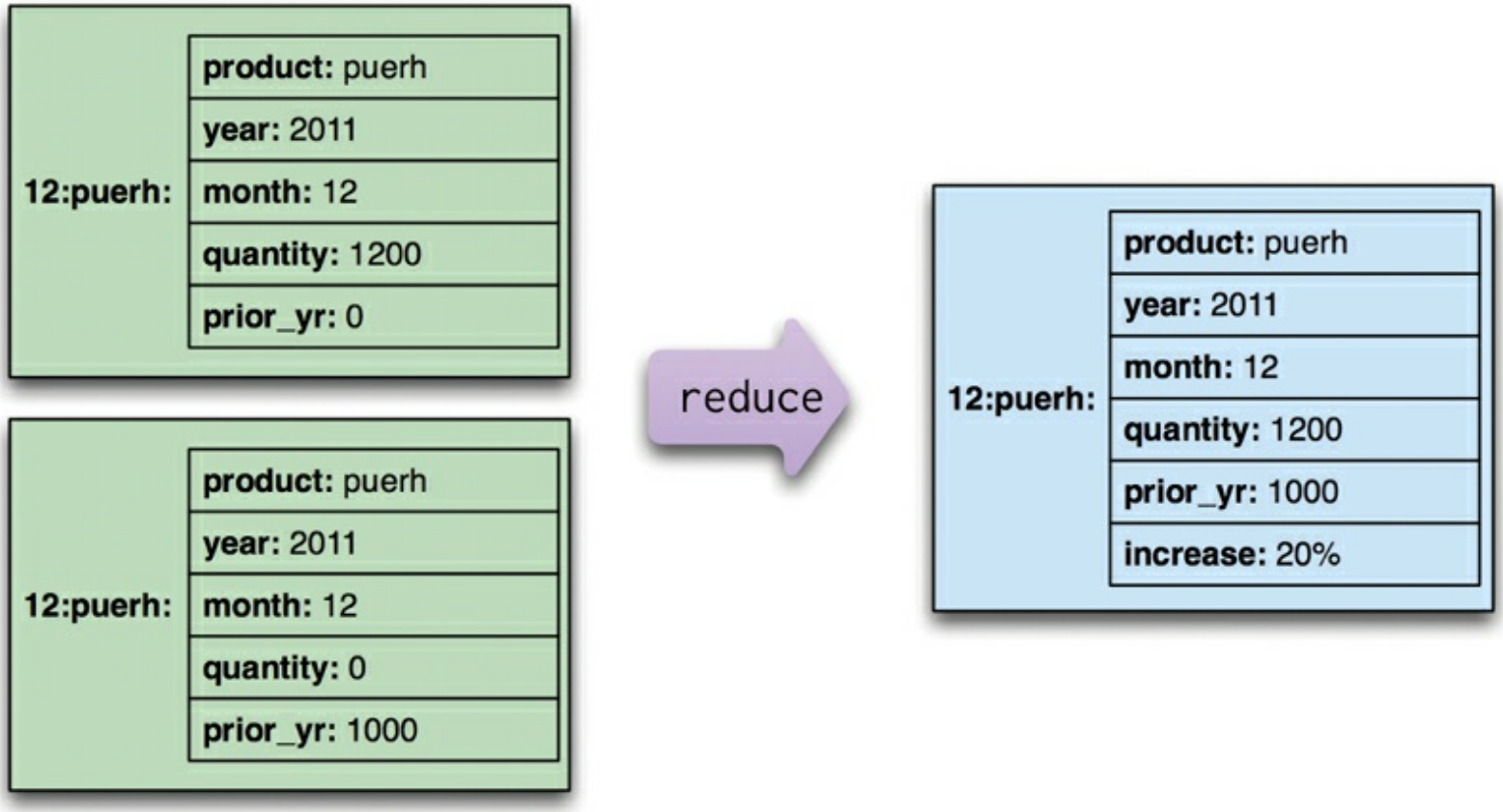

Figure 7.11. The reduction step is a merge of incomplete records.

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

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

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

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

7.3.2. Incremental Map-Reduce

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

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

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

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

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

7.4. Further Reading

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

7.5. Key Points

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