**UNIT-II**

**NOSQL DATA MANAGEMENT**

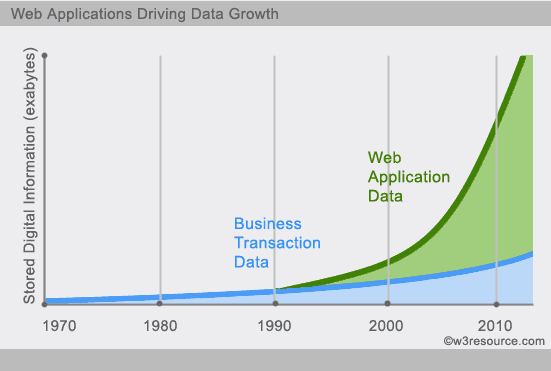
**2.1. Introduction to NoSQL**

**What is NoSQL?**

NoSQL is a non-relational database management systems, different from traditional relational database management systems in some significant ways. It is designed for distributed data stores where very large scale of data storing needs (for example Google or Facebook which collects terabyts of data every day for their users). These type of data storing may not require fixed schema, avoid join operations and typically scale horizontally.

**Why NoSQL?**

In today’s time data is becoming easier to access and capture through third parties such as Facebook, Google+ and others. Personal user information, social graphs, geo location data, user-generated content and machine logging data are just a few examples where the data has been increasing exponentially. To avail the above service properly, it is required to process huge amount of data. Which SQL databases were never designed. The evolution of NoSql databases is to handle these huge data properly.



**NoSQL Data Model**

* It moves away from the relational data model
* Each NoSQL database has a different model
  + Key-value,
  + Document,
  + Column-family,
  + Graph, and
  + Sparse (Index based)
* Of these, the first three share a common characteristic (Aggregate Orientation).

**RDBMS vs NoSQL**

**RDBMS**

- Structured and organized data

- Structured query language (SQL)

- Data and its relationships are stored in separate tables.

- Data Manipulation Language, Data Definition Language

- Tight Consistency

- BASE Transaction

**NoSQL**

- Stands for Not Only SQL

- No declarative query language

- No predefined schema

- Key-Value pair storage, Column Store, Document Store, Graph databases

- Eventual consistency rather ACID property

- Unstructured and unpredictable data

- CAP Theorem

- Prioritizes high performance, high availability and scalability

**2.2. Aggregates**

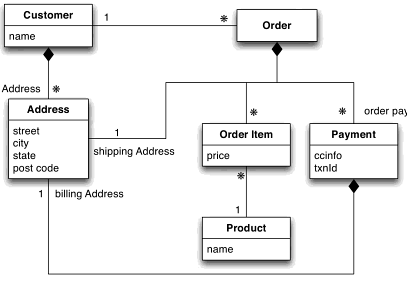
* Aggregate is a term coming from Domain-Driven Design
  + An aggregate is a collection of related objects that we wish to treat as a unit for data manipulation, management a consistency.
  + example: complex record with: simple fields, arrays, records nested inside
* Aggregate is updated with atomic operation
* We like to communicate with our data storage in terms of aggregates
* This definition matches really with how key-value, document, and column-family databases works.
* With aggregates we can easier work on a cluster, since they are unit for replication and sharding.
* Data as units that have a complex structure
  + more structure than just a set of tuples
* Advantages of aggregates:
  + easier for application programmers to work with
  + easier for database systems to handle operating on a cluster

**2.3. Aggregate Data Models**

A data model is the model through which we perceive and manipulate our data. For people using a database, the data model describes how we interact with the data in the database. This is distinct from a storage model, which describes how the database stores and manipulates the data internally. In an ideal world, we should be ignorant of the storage model, but in practice we need at least some inkling of it—primarily to achieve decent performance.

* It recognizes that, you want to operate on data unit having a more complex structure than a set of tuples.
* It think on term of complex record that allows:
  + List,
  + Map,
  + And other data structures to be nested inside it
* Key-Value, document, and column-family databases uses this complex structure.
* Aggregates are also easier for application programmer to work since solve the impedance mismatch problem of relational databases.

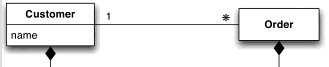
**Example of Aggregate Model**



Here we have two aggregates:

* + Customers and
  + Orders

The link between customer and the order is a relationship between aggregates



**2.4. Key-value and document data models**

* Key-value and Document databases are strongly aggregate-oriented.
* Both of these types of databases consists of lot of aggregates with a key used to get the data.
* The two type of databases differ in that:
  + In a key-value stores the aggregate is opaque (Blob)
  + In a document database we can see a structure in the aggregate.
* The advantage of key-value is that we can store any type of object
* The database may impose some size limit, but we have freedom
* A document store imposes limits on what we can place in it, defining a structure on the data.
  + In return we provide a language to query documents.
* With a key-value we can only access by its key
* With document:
  + We can submit queries based on fields,
  + We can retrieve part of the aggregate, and
  + The database can create index based on the fields of the aggregate.
* But in practice they are used differently
* People use document as key-value
* Riak (key-value) allows you to add metadata to aggregates for indexing
* Redis allows you to break aggregates into lists, sets or maps.
* You can support queries by integrating search tools like Solr. (Riak include solr for searching data stored as XML or JSON).

Despite this the general distinction still holds.

* With key-value databases we expect aggregates using a key
* With document databases, we mostly expect to submit some form of query on the internal structure of the documents.

**2.5. Relationships**

* Relationship between different aggregates:
  + Put the ID of one aggregate within the data of the other
  + Join: write a program that uses the ID to link data
  + The database is ignorant of the relationship in the data

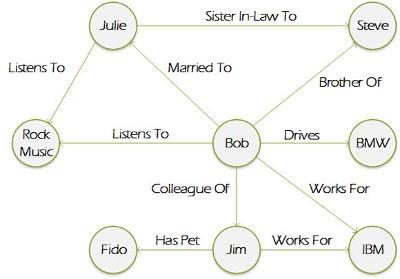
**Relationship management**

* Many NoSQL databases provide ways to make relationships visible to the database
  + Document stores makes use of indexes
  + Riak (key-value store) allows you to put link information in metadata
* But what about updates?
  + Aggregate-oriented databases treat the aggregate as the unit of dataretrieval.
  + Atomicity is only supported within the contents of a single aggregate.
  + Updates over multiple aggregates at once is a programmer's responsibility!
  + In contrast, relational databases provide ACID guarantees while altering many rows through transactions

**2.6. Graph databases**

A graph database, also called a graph-oriented database, is a type of NoSQL database that uses graph theory to store, map and query relationships.

A graph database is essentially a collection of nodes and edges. Each node represents an entity (such as a person or business) and each edge represents a connection or relationship between two nodes. Every node in a graph database is defined by a unique identifier, a set of outgoing edges and/or incoming edges and a set of properties expressed as key/value pairs. Each edge is defined by a unique identifier, a starting-place and/or ending-place node and a set of properties. The mantra of graph database enthusiasts is "If you can whiteboard it, you can graph it."



Graph databases are well-suited for analyzing interconnections, which is why there has been a lot of interest in using graph databases to mine data from social media. Graph databases are also useful for working with data in business disciplines that involve complex relationships and dynamic schema, such as supply chain management, identifying the source of an IP telephony issue and creating "customers who bought this also looked at..." recommendations.

The concept behind graphing a database is often credited to 18th century mathematician Leonhard Euler.

**2.7. Schema less databases**

* No fixed schema:
  + key-value store allows you to store any data you like under a key
  + document databases make 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

**Pros and cons of schema less data**

* Pros:
  + More freedom and flexibility
  + you can easily change your data organization
  + you can deal with non-uniform data
* Cons:
  + A program that accesses data:
    - almost always relies on some form of implicit schema
    - it assumes that certain fields are present
    - carry data with a certain meaning
* The implicit schema is shifted into the application code that accesses data
  + To understand what data is present you have look at the application code
* The schema cannot be used to:
  + decide how to store and retrieve data efficiently
  + ensure data consistency
* Problems if multiple applications, developed by different people, access the same database.
* Relational schemas can be changed at any time with standard SQL commands!

**2.8. Materialized views**

A materialized view is a database object that contains the results of a query. For example, it may be a local copy of data located remotely, or may be a subset of the rows and/or columns of a table or join result, or may be a summary using an aggregate function.

The process of creating a materialized view is sometimes called materialization. This is a form of caching the results of a query, similar to memoization the value of a function in functional languages, and it is sometimes described as a form of precomputation. As with other forms of precomputation, materialized views are typically created for performance reasons, i.e. as a form of optimization.

Materialized views, which store data based on remote tables, are also known as snapshots. A snapshot can be redefined as a materialized view. According to C. J. Date, the term "materialized view" is deprecated in favor of "snapshot".

In any database management system following the relational model, a view is a virtual table representing the result of a database query. Whenever a query or an update addresses an ordinary view's virtual table, the DBMS converts these into queries or updates against the underlying base tables. A materialized view takes a different approach in which the query result is cached as a concrete table that may be updated from the original base tables from time to time. This enables much more efficient access, at the cost of some data being potentially out-of-date. It is most useful in data warehousing scenarios, where frequent queries of the actual base tables can be expensive.

In a materialized view, indexes can be built on any column. In contrast, in a normal view, it's typically only possible to exploit indexes on columns that come directly from (or have a mapping to) indexed columns in the base tables; often this functionality is not offered at all.

**2.9. Distribution Models**

Aggregate oriented databases make distribution of data easier, since the distribution mechanism has to move the aggregate and not have to worry about related data, as all the related data is contained in the aggregate. The primary driver of interest in NoSQL has been its ability to run databases on a large cluster. As data volumes increase, it becomes more difficult and expensive to scale up—buy a bigger server to run the database on. A more appealing option is to scale out—run the database on a cluster of servers. Aggregate orientation fits well with scaling out because the aggregate is a natural unit to use for distribution.

Depending on your distribution model, you can get a data store that will give you the ability to handle larger quantities of data, the ability to process a greater read or write traffic, or more availability in the face of network slowdowns or breakages.

**CAP theorem:**

In a distributed system, managing consistency(C), availability (A) and partition toleration (P) is important, Eric Brewer put forth the CAP theorem which states that in any distributed system we can choose only two of consistency, availability or partition tolerance. Many NoSQL databases try to provide options where the developer has choices where they can tune the database as per their needs. For example if you consider Riak a distributed key-value database. There are essentially three variables r, w, n where

* r=number of nodes that should respond to a read request before it’s considered successful.
* w=number of nodes that should respond to a write request before it’s considered successful.
* n=number of nodes where the data is replicated aka replication factor.

In a Riak cluster with 5 nodes, we can tweak the r,w,n values to make the system very consistent by setting r=5 and w=5 but now we have made the cluster susceptible to network partitions since any write will not be considered successful when any node is not responding. We can make the same cluster highly available for writes or reads by setting r=1 and w=1 but now consistency can be compromised since some nodes may not have the latest copy of the data. The CAP theorem states that if you get a network partition, you have to trade off availability of data versus consistency of data. Durability can also be traded off against latency, particularly if you want to survive failures with replicated data.

NoSQL databases provide developers lot of options to choose from and fine tune the system to their specific requirements. Understanding the requirements of how the data is going to be consumed by the system, questions such as is it read heavy vs write heavy, is there a need to query data with random query parameters, will the system be able handle inconsistent data.

Understanding these requirements becomes much more important, for long we have been used to the default of RDBMS which comes with a standard set of features no matter which product is chosen and there is no possibility of choosing some features over other. The availability of choice in NoSQL databases, is both good and bad at the same time. Good because now we have choice to design the system according to the requirements. Bad because now you have a choice and we have to make a good choice based on requirements and there is a chance where the same database product may be used properly or not used properly.

**2.10. Sharding**

* Different parts of the data onto different servers
  + Horizontal scalability
  + Ideal case: different users all talking to different server nodes
  + Data accessed together on the same node ̶ aggregate unit!
* Pros: it can improve both reads and writes
* Cons: Clusters use less reliable machines ̶ resilience decreases
* Main rules of sharding:

1. Place the data close to where it’s accessed

* Orders for Boston: data in your eastern US data center

1. Try to keep the load even

* All nodes should get equal amounts of the load

1. Put together aggregates that may be read in sequence

* Same order, same node
* Many NoSQL databases offer auto-sharding
* The database takes on the responsibility of sharding

**2.11.** **Master-slave replication**



* Master
  + is the authoritative source for the data
  + is responsible for processing any updates to that data
  + can be appointed manually or automatically
* Slaves
  + A replication process synchronizes the slaves with the master
  + After a failure of the master, a slave can be appointed as new master very quickly
* Pros
  + More read requests:
    - Add more slave nodes
    - Ensure that all read requests are routed to the slaves
  + Should the master fail, the slaves can still handle read requests
  + Good for datasets with a read-intensive dataset
* Cons
  + The master is a bottleneck
    - Limited by its ability to process updates and to pass those updates on
    - Its failure does eliminate the ability to handle writes until:
      * the master is restored or
      * a new master is appointed
  + Inconsistency due to slow propagation of changes to the slaves
  + Bad for datasets with heavy write traffic

**2.12.** **Peer-Peer Replication**



* All the replicas have equal weight, they can all accept writes
* The loss of any of them doesn’t prevent access to the data store.
* Pros:
  + you can ride over node failures without losing access to data
  + you can easily add nodes to improve your performance
* Cons:
  + Inconsistency!
    - Slow propagation of changes to copies on different nodes
      * Inconsistencies on read lead to problems but are relatively transient
    - Two people can update different copies of the same record stored on different nodes at the same time - a write-write conflict.
      * Inconsistent writes are forever.

**2.13.** **Sharding and Replication on MS**



* We have multiple masters, but each data only has a single master.
* Two schemes:
  + A node can be a master for some data and slaves for others
  + Nodes are dedicated for master or slave duties

**Sharding and Replication on P2P**



* Usually each shard is present on three nodes
* A common strategy for column-family databases

**2.14.** **Consistency**

* Write-write conflicts occur when two clients try to write the same data at the same time. Read-write conflicts occur when one client reads inconsistent data in the middle of another client's write.
* Pessimistic approaches lock data records to prevent conflicts. Optimistic approaches detect conflicts and fix them.
* Distributed systems see read-write conflicts due to some nodes having received updates while other nodes have not. Eventual consistency means that at some point the system will become consistent once all the writes have propagated to all the nodes.
* Clients usually want read-your-writes consistency, which means a client can write and then immediately read the new value. This can be difficult if the read and the write happen on different nodes.
* To get good consistency, you need to involve many nodes in data operations, but this increases latency. So you often have to trade off consistency versus latency.
* The CAP theorem states that if you get a network partition, you have to trade off availability of data versus consistency.
* Durability can also be traded off against latency, particularly if you want to survive failures with replicated data.
* You do not need to contact all replicants to preserve strong consistency with replication; you just need a large enough quorum.

**2.15. Relaxing Consistency**

Some different consistency models can be defined by relaxing one or more requirements in sequential consistency called relaxed consistency models. These consistency models do not provide memory consistency at the hardware level. In fact, the programmers are responsible for implementing the memory consistency by applying synchronization techniques.

There are four comparisons to define the relaxed consistency:

**Relaxation:** One way to categorize the relaxed consistency is to define which sequential consistency requirements are relaxed. We can have less strict models by relaxing either program order or write atomicity requirements defined by Adve and Gharachorloo, 1996. Program order guarantees that each process issues a memory request ordered by its program and write atomicity defines that memory requests are serviced based on the order of a single FIFO queue. In relaxing program order, any or all the ordering of operation pairs, write-after-write, read-after-write, or read/write-after-read, can be relaxed. In the relaxed write atomicity model, a process can view its own writes before any other processors.

**Synchronizing vs. Non-Synchronizing:** A synchronizing model can be defined by dividing the memory accesses into two groups and assigning different consistency restrictions to each group considering that one group can have a weak consistency model while the other one needs a more restrict consistency model. In contrast, a non-synchronizing Model assigns the same consistency model to the memory access types.

**Issue vs. View-Based:** Issue method provides sequential consistency simulation by defining the restrictions for processes to issue memory operations. Whereas, view method describes the visibility restrictions on the events order for processes.

**Relative Model Strength:** Some consistency models are more restrict than others. In other words, strict consistency models enforce more constraints as consistency requirements. The strength of a model can be defined by the program order or atomicity relaxations and the strength of models can also be compared. Some models are directly related if they apply same relaxations or more. On the other hand, the models that relax different requirements are not directly related.

The following models are some models of relaxed consistency:

**Weak Ordering:**

There are two categories of memory operations in weak ordering; data operations and synchronization operations.

**Release Consistency**

**Processor Consistency:**

The processor consistency model is similar to PRAM consistency model with a stronger condition that defines all writes to the same memory location must be seen in the same sequential order by all other processes.

**2.16.** **Version Stamps**

Many critics of NoSQL databases focus on the lack of support for transactions. Transactions are a useful tool that helps programmers support consistency. One reason why many NoSQL proponents worry less about a lack of transactions is that aggregate-oriented NoSQL databases do support atomic updates within an aggregate—and aggregates are designed so that their data forms a natural unit of update. That said, it’s true that transactional needs are something to take into account when you decide what database to use.

As part of this, it’s important to remember that transactions have limitations. Even within a transactional system we still have to deal with updates that require human intervention and usually cannot be run within transactions because they would involve holding a transaction open for too long. We can cope with these using version stamps—which turn out to be handy in other situations as well, particularly as we move away from the single-server distribution model.

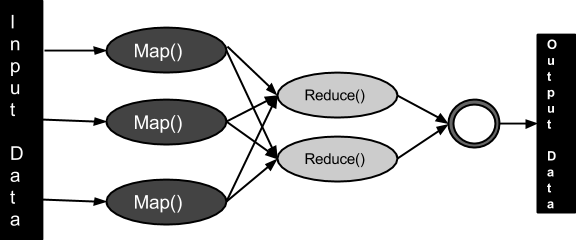
**2.17.** **Map-Reduce**

MapReduce is a processing technique and a program model for distributed computing based on java. The MapReduce algorithm contains two important tasks, namely Map and Reduce. Map takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs). Secondly, reduce task, which takes the output from a map as an input and combines those data tuples into a smaller set of tuples. As the sequence of the name MapReduce implies, the reduce task is always performed after the map job.

The major advantage of MapReduce is that it is easy to scale data processing over multiple computing nodes. Under the MapReduce model, the data processing primitives are called mappers and reducers. Decomposing a data processing application into mappers and reducers is sometimes nontrivial. But, once we write an application in the MapReduce form, scaling the application to run over hundreds, thousands, or even tens of thousands of machines in a cluster is merely a configuration change. This simple scalability is what has attracted many programmers to use the MapReduce model.

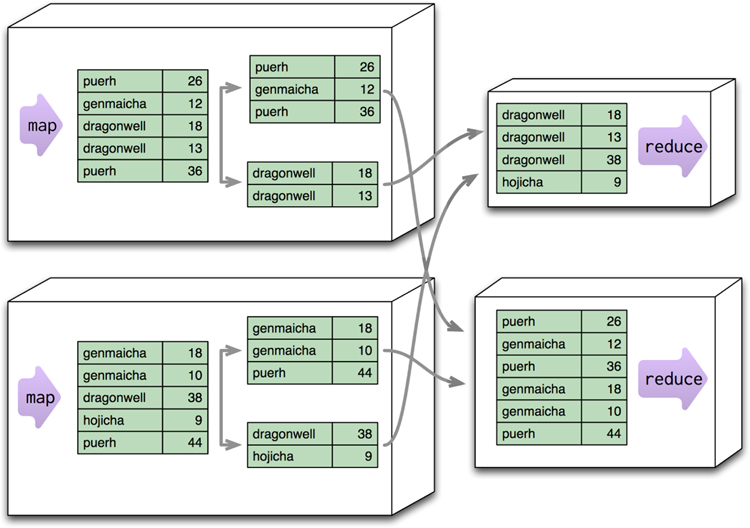
**The Algorithm**

* Generally MapReduce paradigm is based on sending the computer to where the data resides!
* MapReduce program executes in three stages, namely map stage, shuffle stage, and reduce stage.
  + Map stage: The map or mapper’s job is to process the input data. Generally the input data is in the form of file or directory and is stored in the Hadoop file system (HDFS). The input file is passed to the mapper function line by line. The mapper processes the data and creates several small chunks of data.
  + Reduce stage: This stage is the combination of the Shuffle stage and the Reduce stage. The Reducer’s job is to process the data that comes from the mapper. After processing, it produces a new set of output, which will be stored in the HDFS.
* During a MapReduce job, Hadoop sends the Map and Reduce tasks to the appropriate servers in the cluster.
* The framework manages all the details of data-passing such as issuing tasks, verifying task completion, and copying data around the cluster between the nodes.
* Most of the computing takes place on nodes with data on local disks that reduces the network traffic.
* After completion of the given tasks, the cluster collects and reduces the data to form an appropriate result, and sends it back to the Hadoop server.



**2.18.** **Partitioning and Combining**

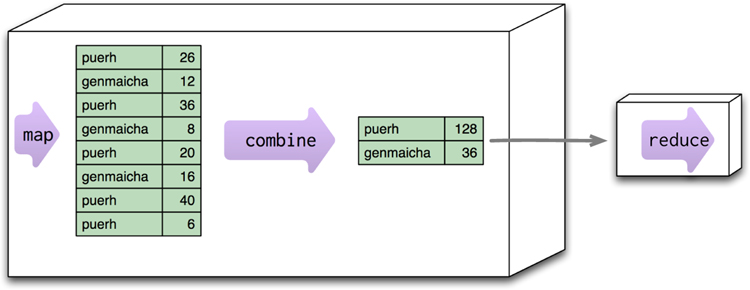
In the simplest form, we think of a map-reduce job as having a single reduce function. The outputs from all the map tasks running on the various nodes are concatenated together and sent into the reduce. While this will work, there are things we can do to increase the parallelism and to reduce the data transfer (see Figure 2.18.1).



**Figure 2.18.1. Partitioning allows reduce functions to run in parallel on different keys.**

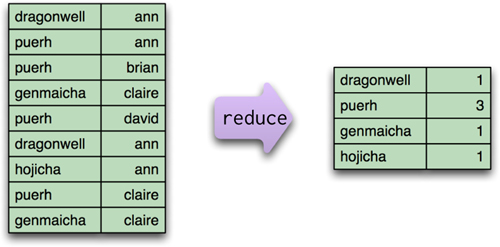
The first thing we can do is increase parallelism by partitioning the output of the mappers. Each reduce function operates on the results of a single key. This is a limitation—it means you can’t do anything in the reduce that operates across keys—but it’s also a benefit in that it allows you to run multiple 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 2.18.2). 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 2.18.2. 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 2.18.3). 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.



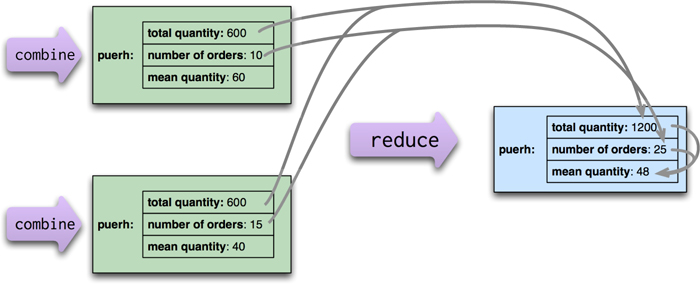
**Figure 2.18.3. This reduce function, which counts how many unique customers order a particular tea, is not combinable.**

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.

**2.19. Composing Map-Reduce Calculations**

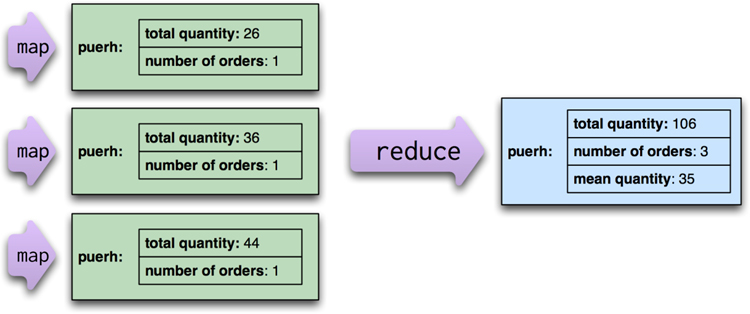
The map-reduce approach is a way of thinking about concurrent processing that trades off flexibility in how you structure your computation for a relatively straightforward model for parallelizing the computation over a cluster. Since it’s a tradeoff, there are constraints on what you can do in your calculations. 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 2.19.1).



**Figure2.19.1. 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 2.19.2).



**Figure 2.19.2. When making a count, each map emits 1, which can be summed to get a total.**