**ACID**

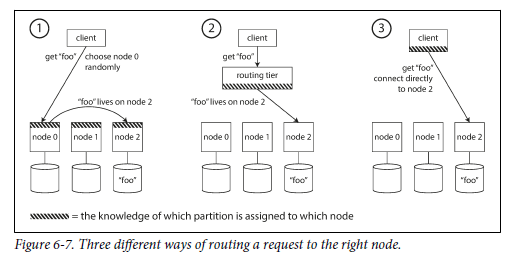
Atomicity: Database transactions, like atoms, can be broken down into smaller parts. When it comes to your database, atomicity refers to the integrity of the entire database transaction, not just a component of it. In other words, if one part of a transaction doesn’t work like it’s supposed to, the other will fail as a result—and vice versa. For example, if you’re shopping on an e-commerce site, you must have an item in your cart in order to pay for it. What you can’t do is pay for something that’s not in your cart. (You can add something into your cart and not pay for it, but that database transaction won’t be complete, and thus not ‘atomic’, until you pay for it).

Consistency: For any database to operate as it’s intended to operate, it must follow the appropriate data validation rules. Thus, consistency means that only data which follows those rules is permitted to be written to the database. If a transaction occurs and results in data that does not follow the rules of the database, it will be ‘rolled back’ to a previous iteration of itself (or ‘state’) which complies with the rules. On the other hand, following a successful transaction, new data will be added to the database and the resulting state will be consistent with existing rules.

Isolation: It’s safe to say that at any given time on Amazon, there is far more than one transaction occurring on the platform. In fact, an incredibly huge amount of database transactions are occurring simultaneously. For a database, isolation refers to the ability to concurrently process multiple transactions in a way that one does not affect another. So, imagine you and your neighbor are both trying to buy something from the same e-commerce platform at the same time. There are 10 items for sale: your neighbor wants five and you want six. Isolation means that one of those transactions would be completed ahead of the other one. In other words, if your neighbor clicked first, they will get five items, and only five items will be remaining in stock. So you will only get to buy five items. If you clicked first, you will get the six items you want, and they will only get four. Thus, isolation ensures that eleven items aren’t sold when only ten exist.

Durability: All technology fails from time to time… the goal is to make those failures invisible to the end-user. In databases that possess durability, data is saved once a transaction is completed, even if a power outage or system failure occurs. Imagine you’re buying in-demand concert tickets on a site similar to Ticketmaster.com. Right when tickets go on sale, you’re ready to make a purchase. After being stuck in the digital waiting room for some time, you’re finally able to add those tickets to your cart. You then make the purchase and get your confirmation. However if that database lacks durability, even after your ticket purchase was confirmed, if the database suffers a failure incident your transaction would still be lost! As you might expect, this is a really bad thing to happen for an online e-commerce site, so transaction durability is a must-have.

**Routing**



**In the harsh reality of data systems, many things can go wrong:**

* The database software or hardware may fail at any time (including in the middle of a write operation)
* The application may crash at any time (including halfway through a series of
* operations)
* Interruptions in the network can unexpectedly cut off the application from the database, or one database node from another
* Several clients may write to the database at the same time, overwriting each other’s changes
* A client may read data that doesn’t make sense because it has only partially been updated.
* Race conditions between clients can cause surprising bugs.

**Issues with Distributed Systems**

Detecting Faults

* A load balancer needs to stop sending requests to a node that is dead (i.e., take it out of rotation).
* In a distributed database with single-leader replication, if the leader fails, one of the followers needs to be promoted to be the new leader

**Slow Database Query**

* Turn on slow query log
* Use Explain keyword
* Index Data
* Remove indexes you don’t need
* Check the WHERE Clause:
  + The WHERE clause connects two or more columns from different tables. This sets up a temporary composite table of data that satisfies the join condition.
  + Here’s the tip: The WHERE clause and related table joins are vital for filtering the elements of your SQL query. If there’s a problem, check this clause first. The problem could be that you’re joining tables on unindexed table fields. That will bog things down because the system will check every record in the tables your referenced instead of using the index to find specific records.
  + One more tip: The fastest results come when the WHERE clause uses indexed table fields.
* Look at the big picture of all data requests
* Remove SELECT ALL if not needed

**Replication**

* Set up Leaders and Followers

**Partitioning**

If every node takes a fair share, then—in theory—10 nodes should be able to handle 10 times as much data and 10 times the read and write throughput of a single node (ignoring replication for now). If the partitioning is unfair, so that some partitions have more data or queries than others, we call it skewed.

Key Value Data

* However, the downside of key range partitioning is that certain access patterns can lead to hot spots. If the key is a timestamp, then the partitions correspond to ranges of time—e.g., one partition per day. Unfortunately, because we write data from the sensors to the database as the measurements happen, all the writes end up going to the same partition (the one for today), so that partition can be overloaded with writes while others sit idle
* To avoid this problem in the sensor database, you need to use something other than the timestamp as the first element of the key. For example, you could prefix each timestamp with the sensor name so that the partitioning is first by sensor name and then by time.

