Part III. Derived Data

In Parts I and II of this book, we assembled from the ground up all the major considerations that go into a distributed database, from the layout of data on disk all the way to the limits of distributed consistency in the presence of faults. However, this discussion assumed that there was only one database in the application.

In reality, data systems are often more complex. In a large application you often need to be able to access and process data in many different ways, and there is no one database that can satisfy all those different needs simultaneously. Applications thus commonly use a combination of several different datastores, indexes, caches, analytics systems, etc. and implement mechanisms for moving data from one store to another.

In this final part of the book, we will examine the issues around integrating multiple different data systems, potentially with different data models and optimized for different access patterns, into one coherent application architecture. This aspect of system-building is often overlooked by vendors who claim that their product can satisfy all your needs. In reality, integrating disparate systems is one of the most important things that needs to be done in a nontrivial application.

Systems of Record and Derived Data

On a high level, systems that store and process data can be grouped into two broad categories:

Systems of record

A system of record, also known as *source of truth*, holds the authoritative version of your data. When new data comes in, e.g., as user input, it is first written here. Each fact is represented exactly once (the representation is typically *normalized*). If there is any discrepancy between another system and the system of record, then the value in the system of record is (by definition) the correct one.

Derived data systems

Data in a derived system is the result of taking some existing data from another system and transforming or processing it in some way. If you lose derived data, you can recreate it from the original source. A classic example is a cache: data can be served from the cache if present, but if the cache doesn't contain what you need, you can fall back to the underlying database. Denormalized values, indexes, and materialized views also fall into this category. In recommendation systems, predictive summary data is often derived from usage logs.

Technically speaking, derived data is *redundant*, in the sense that it duplicates existing information. However, it is often essential for getting good performance on read queries. It is commonly *denormalized*. You can derive several different datasets from a single source, enabling you to look at the data from different "points of view."

Not all systems make a clear distinction between systems of record and derived data in their architecture, but it's a very helpful distinction to make, because it clarifies the dataflow through your system: it makes explicit which parts of the system have which inputs and which outputs, and how they depend on each other.

Most databases, storage engines, and query languages are not inherently either a system of record or a derived system. A database is just a tool: how you use it is up to you. The distinction between system of record and derived data system depends not on the tool, but on how you use it in your application.

By being clear about which data is derived from which other data, you can bring clarity to an otherwise confusing system architecture. This point will be a running theme throughout this part of the book.

Overview of Chapters

We will start in Chapter 10 by examining batch-oriented dataflow systems such as MapReduce, and see how they give us good tools and principles for building large-scale data systems. In Chapter 11 we will take those ideas and apply them to data streams, which allow us to do the same kinds of things with lower delays. Chapter 12 concludes the book by exploring ideas about how we might use these tools to build reliable, scalable, and maintainable applications in the future.