CHAPTER 9

Fast Data Patterns

In this chapter, we examine well-known patterns in developing fast data applications. As you know, there are two approaches: (1) the batch, on disk, traditional approach and (2) the streaming, on memory, modern approach. The patterns in this chapter apply to both approaches.

The chapter has three main sections. In the first, we discuss the concept of *fast data* to differentiate it from big data. In the second section, we discuss the differences between ACID and CAP in order to understand the capabilities and limitations of both in fast data. The third section features recipes with design patterns to write certain types of streaming applications.

The chapter's goal is to make a cookbook with a recipe collection for fast data application development. Of course, there are many more recipes and patterns than revealed here, but recall that the fast data approach is relatively new.

This chapter covers the following:

- Fast data
- ACID vs. CAP
- Integrating streaming and transactions
- Streaming transformations
- Fault recovery strategies
- Tag data identifiers

Fast Data

Lately, some marketing and IT companies have abused some important terms to create great illusions, which only resulted in frustration when it was discovered that these buzzwords were not the panacea that everyone expected; two of these terms were *big data* and *the cloud*.

If you are perceptive, you may have noticed that in this book we try to avoid the term "big data" because although many organizations require data analysis, they do not have large volumes of data. Businesses do not really want big data, they need fast data.

At this moment, we are living in the fast data explosion, driven by mobile-devices proliferation, the Internet of Things (IoT), and machine-to-machine (M2M) communication. In regards to business needs, it is due to close interaction with customers, personalized offers, and reaction recording.

One characteristic of fast data applications is the ingestion of vast amounts of data streams. Note the *big* difference between ingestion and storage. Businesses require real-time analysis and the need to combine transactions on live data with real-time analytics.

R. Estrada and I. Ruiz, *Big Data SMACK*, DOI 10.1007/978-1-4842-2175-4_9

Fast data applications solve three challenges:

- Data streams analysis
- Data pipelines
- Real-time user interaction

We are too close to this change, so we cannot accurately distinguish the border between big data and fast data. Nor can we precisely identify which one has the greater value for business. All we know so far is that each one brings different values.

Another distinguishing phenomenon is that we have reached the boundaries of traditional models. For example, we consider the model of relational databases a pillar of all modern technological knowledge. We have reached the level where questioning relational model transitions is no longer a far-fetched proposal but a viable recurring option.

NoSQL solutions offer speed and scale in exchange for a lack on transactionality and query capabilities. Today, developers do not use a single technology; they have to use several (a clear example is the SMACK stack) because one technology is no longer enough. The problem with this approach is that it has a steep learning curve, often adds unnecessary complexity, causes duplication of effort, and often sacrifices performance to increase speed.

The question to answer in this chapter (and in the book) is this: How do we combine real-time data stream analysis with a reliable, scalable, and simple architecture?

Many companies have opted for the traditional batch approach, but history has shown that it requires too much infrastructure and both human and computational efforts. Or we can opt for a modern approach, which often involves the challenge of traditional paradigms, such as Batch, SQL, and ACID processing. Although there are many skeptics, this approach simplifies development and increases the performance by reducing infrastructure costs.

Fast Data at a Glance

Today's world is interactive. Information delivery should go to the right person at the right device in the right place at the right moment; or using the correct terms—personalized, ubiquitous, geolocalized, and in real time. That is what you call fast data.

However, building fast data applications requires a tremendous skill set. This chapter is a compendium of some patterns to handle analysis of data streams with operational workloads. A pattern is a recipe; this chapter is a cookbook to overcome the well-known challenges with new and more predictable applications.

Fast data applications must scale across multiple machines and multiple coordinate systems, and above all, reduce the complexity of the issue. Recall that an application must be simple, reliable, and extensible. In order to quickly implement data, you need to understand the structure, the data flow, and the implicit requirements of data management.

Right now fast data styles are being created for developers who are having problems with current development scalability. Many fast data issues far exceed the capabilities of traditional tools and techniques, creating new challenges still unresolved by systems that are slow and don't scale.

Modern problems cannot be solved by traditional approaches. The new tools must be created from thinking differently and using approaches that challenge traditional paradigms. That's how LinkedIn generated Kafka, Facebook generated Cassandra, and the AMPLab generated Spark and Mesos. And in turn, this generated new companies, such as Confluent, DataStax, Databricks, and Mesosphere.

If anything is certain it is that when each technology in the SMACK stack was coded, the thinking and skills of the people involved were vastly different from what they gained from past experiences.

Beyond Big Data

In a world with abundant and mundane discussions on big data, where marketing makes more noise than technology, fast data was born in a work context, midnight calls, and aggressive and excessive competition between companies looking to provide the best service at the lowest cost. Fast data is the agent of change; the engine that defines a new economy.

In a nutshell, you can say that fast data is data on the move. It is the streaming of hundreds of millions of endpoints to applications and servers. Imagine if you can mobile devices, sensors, financial transactions, logs, retail systems, telecommunication routers, authorization systems, and so forth. Everything changes for developers; you can only say that the increase in data is constant. There is Moore's law, which states that each year the amount of data is doubled. As mentioned in earlier chapters, the world was a quiet and peaceful place when data was stored for eternal rest; that's what you call big data, which is stored in Hadoop, in data warehouses, and in *data lakes*.

Fast data, on the other hand, is data arising from turmoil, data in motion, data streaming. An intrinsic feature of fast data is that data streams have to be treated in real time. The big data era is based on the analysis of structured and non-structured data stored in Hadoop and data warehouses through batch processes.

The SMACK stack emerges across verticals to help developers build applications to process fast data streams (note that here you also use the term *fast data stream* instead of *big data stream*). The sole purpose of the SMACK stack is processing data in real time and outputting data analysis in the shortest possible time, which is usually in milliseconds. For example, bank authorizations on credit cards can't delay too long before the client application times out. Although, some applications tolerate responses on minutes, the scenario where the big data analysis is delivered tomorrow is no longer viable.

Fast Data Characteristics

Fast data applications meet several characteristics. As we will discuss later, they influence architecture decisions. There are three main characteristics:

- Fast ingestion
- Analysis streaming
- Per event transactions

Fast Ingestion

The first stage in the data streaming process is data ingestion. The purpose of ingestion is to have a direct interface with data sources in order to make changes and the normalization of input data. Ingestion presents the challenge of extracting value from data and labeling it by assigning key-value pairs.

There are two types of ingestion:

- **Direct ingestion**. Here a system module hooks directly with the API generation. System speed depends on the speed of the API and the network. The analysis engines have a direct adapter. One advantage is that it can be very simple; a disadvantage is that it is not flexible—making changes that don't affect the performance can be a complex process.
- **Message queue**. When you do not have access to the data generation API, you can use a broker such as Apache Kafka. In this case, the data is managed in the form of queues. The advantage, as you know, is that you can partition, replicate, sort, and manage the pipeline in proportion to the pressure over the slower component.

Analysis Streaming

As the data is created, it reaches the analysis engine. Data can come in multiple types and formats. Often the data is enriched with metadata about the transformation process. This information may come through messages or events. Examples include sensor data of all types, user interaction with a web site, transaction data, and so forth.

The increase in the amount of fast data has made analysis move from the backend layer to the streaming layer. Every day there is less analysis in data warehouses. The ability to analyze data streams and make decisions with live transaction is most wanted today.

Per Event Transactions

As analysis platforms have to produce real-time analysis over incoming data, analysis speed far exceeds human speed. Hence, machine learning tools on streaming data are recent trending.

To generate value on streaming data, you must take action in real time. This has two reasons. The first is a technical reason related to the fact that real-time data chunks are "stored" in memory, and you cannot store them on a disk or another storage medium, because you quickly flood large amounts of space and because you likely don't have the money and hardware of Facebook or Twitter to indefinitely store the data. The second is a business reason that has to do with decision making in real time; for example, online authorization charges, real-time user interaction with web site recording, real-time multiplayer game engines, and so forth.

The ability to extract information as it arrives and to combine it with business logic in real time makes possible modern fraud detection systems, trading shares on the stock market, or the Uber operation.

On the data management layer, all the actions must be able to read and write many pieces of data, storing only results, inferences, and recommendations. It is worth noting that online input data is not stored, because there is no space or budget to store this amount of data.

All of this can be summarized as the interaction of the event when it arrives. The streams of high-speed data required to have high-availability schemas are discussed later in a section on the at-least-one schema on event delivery.

The modern challenge for data engineers is the extraction and capture of the value on per-event transactions.

Fast Data and Hadoop

And what if you already have our big data model mounted on Apache Hadoop? Well, the important thing here is to build a front end. The front end of a big data system *must* have every one of the following functions: filter, de-dupe, aggregate, enrich, and denormalize.

If you already have a model like Hadoop, it is important to change the whole paradigm. You can continue using Hadoop, but instead of storing all the information in Hadoop as it arrives, you use an engine like Spark to move all Lambda Architecture to a streaming architecture, and from a batch processing model to an online pipeline processing model.

The associated costs and the time to do common operations, as filter, de-dupe, aggregate, and so forth, in a model such as the Spark is drastically reduced compared if you made it over Apache Hadoop with a next-day batch model. Moreover, a batch model usually has no redundancy and high availability schemas, and if so, they are very expensive.

The batch processing schemas always require the process of cleaning data before storing it. In a pipelined architecture, the cleaning process is part of the ingestion process.

Another modern alternative is to dump the Hadoop Distributed File System (HDFS). An advantage of the pipeline model is that very old and obsolete data can be eliminated as new data arrives. In the pipeline model, no garbage is stored because the data stored is not input data, but those produced by the same engine.

A Hadoop developer's recurring complaint is the difficulty of analysis to scale. With a pipeline model, counting and aggregation reduces the problem by several orders of magnitude. By reducing the size of stored data, you reduce the time to analyze it.

Hadoop developers also complain when they have to send aggregates to HDFS; with a fast data front end, this doesn't happen because the aggregates are sent as they arrive—no batch process and everything is microbatching in Spark.

Data Enrichment

Data enrichment is another advantage of fast data over traditional models. The data always has to be filtered, correlated, and enriched before being stored in the database. Performing the enrichment process at the streaming stage provides the following advantages:

- NoETL process. As you saw, unnecessary latency created by ETL processes is avoided in a streaming model.
- Unnecessary disk I/O is removed. As you saw, as Hadoop solutions are based on disk, everything in fast data is based on memory. Everything is in real time because there is no time for batch processes.
- The use of hardware is reduced. Because you don't have to store everything and you don't have to do very complex analysis over data lakes, the cost of hardware is dramatically reduced; resources (processor, memory, network, and disk) are used more efficiently.

Since fast data entry feeds are information streams, maintaining the semantics between streams is simpler because it creates a consistent and clean system. This can only be achieved if you act in each event individually; here there are no big data windows or handling large data chunks susceptible to errors.

These per event transactions need three capacities:

- **Stream connection oriented**. You need clusters of Kafka, Cassandra, Spark, and Hadoop/HDFS.
- **Fast searches**. To enrich each event with metadata.
- **Contextual filtering**. Reassembles discrete input events in logical events that add more meaning to the business.

In short, transactions per event require the entire system to be stateful; that is, everything is in memory and has to store the minimum in disk.

Queries

Take the example of advertising based on user clicks on a given web page. How do you know which ad the user clicked? How do you know that it wasn't a robot? How do you know the amount to charge the advertising agency at the end of month?

Another example is when you have a security system attack. How do you know when you are being attacked by a denial of service? How do you know that an operation is fraudulent? To find out if another machine is attacking, should you consider only the information from the last hour?

Today all contracts are based on a *service-level agreement* (SLA). In order to verify at the end of the month that you meet those contracts, you need to make queries, sometimes very sophisticated, of the data within your system.

Not meeting an SLA could lead to multimillion-dollar sanctions. Knowing that you met the SLA requires the ability to make queries in your system. This fast data feature allows the user to query at any time and over any time frame.

ACID vs. CAP

Fast data is a transformation process. There are two key concepts in modern data management: the ACID properties and CAP theorem. In both acronyms, the C stands for *consistency*, but it means something different to each. We will discuss the differences between the Cs later.

Let's now delve into transactions. The core concepts of a transaction are semantics and guarantees. The more data a computer handles, more important its function, but also more complex and prone to errors.

At the beginning of the computer age, when two computers had to write the same data at the same time, the academia noted that the process should be regulated to ensure that data was not corrupted or written incorrectly. When computers were exposed to human interaction, the risk of human error in the middle of a calculation became a major concern.

The rules were defined by Jim Gray[1] and published by the Association for Computing Machinery (ACM) in 1976. In the 1980s, IBM and other companies were responsible for popularizing ACID. It was like everything in computer science: on paper things worked perfectly, but in practice strong performance discussions are untied. Today, ACID transactions are a mainstay in any database course.

A transaction consists of one or more operations in a linear sequence on the database state. All modern database engines should start, stop, and cancel (or roll back) a set of operations (reads and writes) as a metadata operation.

Transactional semantics alone do not make the transaction. You have to add ACID properties to prevent developers from being lost when they have concurrent access on the same record.

ACID Properties

ACID means Atomic, Consistent, Isolated, and Durable.

- **Atomic**. All the transaction parts should be treated as a single action. This is the mantra: All parts are completed or none is completed. In a nutshell, if part of the transaction fails, the state of the database remains unchanged.
- **Consistent**. Transactions must follow the rules and restrictions defined by the database (e.g. constraints, cascades, triggers). All data that is written to the database must be valid. No transaction must invalidate the database state. (Note that this is different from the C in the CAP theorem.)
- **Isolated**. To achieve concurrency control, transactions isolation must be the same as if you were running the transactions in serial, sequentially. No transaction should affect another transaction. In turn, any incomplete transaction should not affect another transaction.
- **Durable**. Once the transaction is committed, the change must be persisted and should not change anymore. Likewise, it should not cause conflicts with other operations. Note that this has nothing to do with writing to disk and recent controversies, because many modern databases live on memory or are distributed on the users' mobile devices.

[1] http://dl.acm.org/citation.cfm?doid=360363.360369

CAP Theorem

The CAP theorem is a tool to explain the problems of a distributed system. It was presented by Eric Brewer at the 2000 Symposium on Principles of Distributed Computing, and formalized and demonstrated (as good theorem) by Gilbert and Lynch[2] in 2002.

CAP means Consistent, Available, and Partition Tolerant.

- **Consistent**. All replicas of the same data must have the same value across the distributed system.
- **Available**. Each living node in a distributed system must be able to process transactions and respond to queries.
- **Partition Tolerant**. The system will continue to operate even if it has network partitioning.

These are the original sad words of the CAP theorem: "In the face of network partitions, you can't have both perfect consistency and 100% availability. Plan accordingly."

It is a very sad theorem because it does not mention what is possible, but the impossibility of something. The CAP theorem is known as the "You-Pick-Two" theorem; however, you should avoid this conception, because choosing AP does not mean that you will not be consistent, and choosing CP does not mean that you will not be available. In fact, most systems are not any of the three. It means that designing a system is to give preference to two of the three characteristics.

Furthermore, it is not possible to choose CA. There cannot be a distributed system in which the partitions are ignored. By definition, a non-partitioned network means not having a distributed system. And if you don't have a distributed system, the CAP theorem is irrelevant. Thus, you can never exclude the P.

Consistency

The ACID consistency was formulated in terms of databases. Consistency in the CAP theorem is formulated in terms of distributed systems.

In ACID, if a scheme states that a value must be unique, then a consistent system reinforces the uniqueness of that value across all operations. A clear example is when you want to delete a primary key when you have references to other tables using constraints; the database engine will indicate that there are children records and you cannot erase the key.

The CAP consistency indicates that each replica of the original value—spread across the nodes of a distributed system—will always have the same value. Note that this warranty is logical, not physical. Due to network latency (even running over optic fiber at the speed of light), it is physically impossible for a replication of all nodes to take zero seconds. However, the cluster can present a logical view to customers to ensure that everyone sees the same value.

The two concepts reach their splendor when systems offer more than a simple key-value store. When systems offer all ACID properties across the cluster, the CAP theorem makes its appearance, restricting the CAP consistency.

On the other hand, when you have CAP consistency, through repeated readings and full transactions, the ACID consistency should be offered in every node. Thus, the systems that prefer CAP availability over CAP consistency rarely ensure ACID consistency.

[2] http://dl.acm.org/citation.cfm?id=564601:

CRDT

To explain *eventual consistency*, consider the example of a cluster with two machines. When the cluster works, the writings are spread equally to both machines and everything works fine. Now suppose that communication between the two machines fails but the service is still active. At the end of the day, you will have two machines with different information.

To rectify this fault, traditional approaches offer very complex rectification processes to examine both servers and try to resynchronize the state.

Eventual consistency (EC) is a process that facilitates the data administrator's life. The original white paper on Dynamo (the Amazon database)[3] formally defined eventual consistency as the method by which several replicas may become temporarily different, but eventually converge to the same value. Dynamo guarantees that the fix process is not complex.

It is worth noting that eventual consistency is not immediate; so two queries may yield different results until synchronization is complete. The problem is that EC does not guarantee that the data converges to the latest information, but to the more correct value. This is where the correctness definition becomes complicated.

Many techniques have been developed to offer an easier solution under these conditions. It's important to mention *conflict-free replicated data types* (CRDTs). The problem with these methods is that in practice, they offer fewer guarantees on the final status of the system than those offered by the CAP theorem. The benefit of CRDT is that under certain partitioning conditions, the high availability offer leaves nodes operating.

The EC Dynamo-style is very different from the log-based rectification methods offered by the bank industry to move money between bank accounts. Both systems can diverge for a period of time, but banks usually take longer and reach a more precise agreement.

Integrating Streaming and Transactions

Imagine the operation of these high-speed transactional applications: real-time payments, real-time authorization, anti-fraud systems, and intelligent alerting. These applications would not be conceived today if there weren't a mix of real-time analysis and transaction processing.

Transactions in these applications require real-time analysis as input. Since it is impossible in real time to redo the analysis based on data derived from a traditional data store, to scale, you must keep streaming aggregation within the transaction. Unlike regular batch operations, aggregation streams maintain the consistency, up-to-dateness, and accuracy of the analysis, which is necessary for the transaction.

In this pattern, you sacrifice the ability to make ad-hoc analyses in exchange for high-speed access to the analysis, which is necessary for the application.

Pattern 1: Reject Requests Beyond a Threshold

Consider a high-volume-requests web page that implements sophisticated usage metrics for groups and individual users as a function of each operation.

The metrics are used for two main purposes:

- Billing charges based on use
- To force the same contracted service quality level (expressed as the number of requests per second, per user, and per group).

In this case, the platform implementation of the policy verification should have counters for every user and group of users. These counters must be accurate (because they are inputs for billing and implementation of service quality policies), and they must be accessible in real time to evaluate and authorize (or deny) new accesses.

[3] http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

It is necessary to maintain a balance in real time for each user. To accurately maintain a balance, you need an ACID OLTP system. The same system requires the ability to maintain high-speed aggregations. The scalability of the solution is achieved by combining aggregation transactions with real-time high-speed transmission. Examples of these systems include new credits granting systems and used credit deductions).

Pattern 2: Alerting on Predicted Trends Variation

Imagine an operational monitoring platform where you need to issue warnings or alarms when a threshold predicate is exceeded at a statistically significant level. This system combines two capabilities:

- It keeps the analysis on real time (counters, streaming add-ons, and status summary of current use)
- It compares the analysis with the predicted trend. If the trend is exceeded, the system should raise an alert.

The system records this alarm to suppress other alerts (limiting the intermittency of an alarm for a single event). This is another system that requires the combination of analytical and transactional capabilities.

Analyzed separately, this system needs three independent systems working simultaneously:

- An analysis system that is microdosing real-time analysis
- An application reading these analyses and the trend line predicted to generate alerts on the application
- A transactional system that stores the generated alerts data and that implements suppression logic

The execution of three tightly coupled systems like this (our solution requires the three systems running) reduces the reliability and complicates the operation.

To achieve real time analysis you need to combine the request-response system with event processing streaming applications.

When Not to Integrate Streaming and Transactions

OLAP (online analytical processing) systems offer the benefit of rapid analytical queries without pre-aggregation. These systems can execute complex queries over huge data, but within the threshold, they work in batch, reporting to human analysts in the data workflow. These systems are not compatible with high-speed transactional workloads because they are optimized to batch reporting, not OLTP (online transaction processing) applications.

Aggregation Techniques

Pre-aggregation is a technique with many algorithms and features developed. The following are common techniques for implementing real-time aggregation:

- **Windowed events.** Used to express moving averages or a timeframe summary of a continuous event. These techniques are found in CEP (complex event processing) or microbatching systems like Apache Spark.

- **Probabilistic data structures**. Data is added within a certain margin of error bounded by probability. These algorithms typically exchange precision for space, allowing estimation in a smaller storage space. Examples of probabilistic data structures and algorithms include Bloom filters, as in Apache Cassandra.
- **Materialized views**. A view could define aggregation, partition, filter, or join. Materialized views group the base data and keep a physical copy of the resulting data. Materialized views allow declarative aggregations, which eliminate coding and offer easy, concise, and accurate aggregation. You find examples of this in Oracle DB, Postgres, SQL Server, and MySQL.

Streaming Transformations

Effectively processing big data often requires multiple database engines, each with a special purpose. Usually, systems good at online CEP (complex event processing) are not good at batch processing against large data volumes. Some systems are good for high-velocity problems; others are good for large volume problems. In most cases, you need to integrate these systems to run meaningful applications.

Usually, the data arrives at high-velocity ingest-oriented systems and is processed into the volume-oriented systems. In more advanced cases, predictive models, analysis, and reports are generated on the volume-oriented systems and sent as feedback to the velocity-oriented systems to support real-time applications. The real-time analysis from the fast systems is integrated into downstream applications and operational dashboards that process real-time alerts, alarms, insights, and trends.

In a nutshell, many fast data applications run on top of a set of tools. Usually, the platform components include the following:

- At least one large shared storage pool (e.g., HDFS, Apache Cassandra)
- A high performance BI analytics query tool (e.g., a columnar SQL system, Apache Spark)
- A batch processing system (e.g., Map Reduce, Apache Spark)
- A streaming system (e.g. Apache Kafka)

The input data and processing output move across these systems. The key to solve many big data challenges is the design of this dataflow as a processing pipeline that coordinates these different systems.

Pattern 3: Use Streaming Transformations to Avoid ETL

The new events captured into a long-term repository often require transformation, filtering, or processing before being available for exploitation. For example, an application that captures user sessions consisting of several discrete events, enriching those events with static data to avoid expensive repeated joins in the batch layer, and/or filtering redundant events storing only unique values.

There are (at least) two approaches to run this process:

1. The data can be stored in a long-term repository and then ETLed (extracted, transformed, and loaded) to its final form. This approach trades I/O, storage, and time (results are delayed until the entire ETL process is completed) for a slightly simpler architecture (e.g., move data directly from a queue to HDFS).
2. This approach is referred as *schema on read*. It reduces the choice of back-end systems to those of schema-free systems, removing the non-optimal, depending on your specific reporting requirements.

3. Execute the transformations in a streaming way before the data arrives at the long-term repository. This approach adds a streaming component (Apache Spark) between the source queue (Apache Kafka) and the final repository (Apache Cassandra), creating a continuous processing pipeline.

4. Moving the entire process to a real-time processing pipeline has several advantages. Writing I/O to the back-end system is drastically reduced (in the first model, raw data input is written, and then the ETLed data is written; with this approach only ETLed data is written).

5. You have this comparison between the two approaches:

    - The second model leaves more I/O available for the primary purpose of the back-end repository: data analysis and reporting activities.

    - In the second approach, the operational errors are noticeable nearly in real time. When using ETL, the raw data is not inspected until the entire ETL process runs. This delays operational notifications of corrupt inputs or missing data.

    - In the second model, the time to insight is reduced. For example, when managing data of sessions, a session can participate in batch reporting before being closed. With ETL, you must wait approximately half of the ETL period before the data is available for processing.

Pattern 4: Connect Big Data Analytics to Real-Time Stream Processing

The incoming events from real-time processing applications require back-end systems analysis. The following are some examples:

- On an alerting system that notifies when an interval exceeds historical patterns, the data describing the historical pattern needs to be available.

- Applications managing real-time customer experiences or personalization often use customer segmentation reports generated by statistical analysis running on a batch analytics system.

- Hosting OLAP outputs in a fast, scalable query cache to support operators and applications that need high-speed and highly concurrent access to data.

In all of these cases, the reports, analysis, and models from big data analytics need to be made available to real-time applications. In this case, information flows from the big data–oriented system (batch) to the high velocity system (streaming).

This brings important requirements:

- The fast data, speed-oriented application requires a data management system capable of holding the state generated by the batch system.

- This state needs to be regularly updated or fully replaced. There are a few common ways to manage the refresh cycle. The best trade-off depends on the specific application.

Some applications (e.g., applications based on user segmentation) require per record consistency but can tolerate eventual consistency across records. In these cases, updating state in the velocity-oriented database on a per-record base is sufficient.

Updates will need to do the following:

- Communicate new records (in our example, new customers)
- Update existing records (customers that have been recategorized)
- Delete outdated records (ex-customers)

The records in the velocity system should be timestamped for operational monitoring and alerts generated if stale records persist beyond an expected refresh cycle.

Other applications require the analysis to be strictly consistent. If it is not sufficient for each record to be internally consistent, the records set as a whole requires guaranteed consistency. These cases are often seen in analytic query caches. Often these caches are queried for additional levels of aggregation; aggregations that span multiple records. Producing a correct result therefore requires that the full data set be consistent.

A reasonable approach to transferring this report data from the batch analytics system to the real-time system is to write the data into a shadow table. Once the shadow table is completely written, it can be atomically swapped, with the main table addressed by the application. The application will either see only data from the previous version of the report or only data from the new version of the report, but it will never see a mix of data from both reports in a single query.

Pattern 5: Use Loose Coupling to Improve Reliability

When connecting multiple systems, it is imperative that all systems have an independent administration. Any part of the pipeline could fail, although the other systems remain available and functional. If the (batch) back end is offline, the (high velocity) front end should still operate, and vice versa.

This requires thinking through several design constraints:

- The location of data that cannot be pushed (or pulled) through the pipeline. The components responsible for the durability of stalled data.
- The failure and availability model of each component in the pipeline.
- The system recovery process. The list of the components that have record meaning or can have recovery sources for lost data or interrupted processing.
- When a component fails, the list of the components become unavailable. The time upstream components maintain functionality.

These constraints form the *recovery time objective* (RTO).

In every pipeline, there is a slowest component, the *bottleneck*. When designing a system, you must explicitly choose a component that will be the bottleneck. Having many systems, each with identical performance, a minor degradation on any system will create an overall bottleneck. Operationally, this is a pain. It is better to choose the most reliable component as the bottleneck or the most expensive resource as the bottleneck so that you can achieve a more predictable reliability level.

Points to Consider

Connecting multiple systems is always a complex task. This complexity is not linear with the number of systems. Generally, complexity increases in the function of connections (in the worst-case scenario, you have N*N connections between N systems).

This problem cannot be solved with a single stand-alone monolithic system. However, high velocity and large volume data management scenarios typically require a combination of specialized systems. When combining these systems, you must carefully design dataflow and connections between them.

Make a couple of designs, so each system operates independently of the failure of the others. Use multiple systems to simplify; for example, replace batch ETL processes with continuous processes.

Fault Recovery Strategies

Most streaming applications move data across multiple processing stages.

Often, the events land in a queue and are then read by downstream components. Those components might write new data back to a queue as they process, or might directly stream data to their downstream components. Building a reliable data pipeline always implies designing fault recovery strategies.

When multiple processing components are connected, statistically, one component will fail, or become unreachable or unavailable. When this occurs, the other components should continue receiving data. When the failed component comes back online, it must recover its previous state and then continue processing new events. Here we discuss how to resume processing.

Idempotency is a specific technique to achieve exactly-once semantics.

Additionally, when processing horizontally scaled pipelines, each stage has multiple servers or processes running in parallel. A subset of servers within the cluster can always fail. In this case, the failed servers need to be recovered and their work needs to be reassigned.

There are a few factors that complicate these situations and lead to different trade-offs:

- Usually, it is uncertain to determine what the last processed event was. Typically, it is not feasible to two-phase commit the event processing across all pipeline components.
- Event streams are often partitioned across several processors. Processors and upstream sources can fail in arbitrary combinations. Picturing everything as a single, unified event flow is not recommendable.

Here we discuss three options for resuming processing distinguished by how events near the failure time are handled. The approaches to solving the problem are explained as follows.

Pattern 6: At-Most-Once Delivery

At-most-once delivery allows dropping some events. In this case, the events not processed by an interruption are simply dropped and they don't become part of the input of the downstream system. Note that, this pattern is only accepted when the data itself is low value or loses value if it is not immediately processed.

The following are points to evaluate an at-most-once delivery:

- The historical analytics should show which data is unavailable.
- If the event stream will be eventually stored into an OLAP or data lake, you need reports and data algorithms that detect and manage the missing values.
- It should be clear which data is lost.

Lost data generally is not aligned exactly with session boundaries, time windows, or even data sources. It is also probable that only partial data was dropped during the outage period, so some values are present.

Additional points to be considered include the maximum outage period and the size of largest dropped gap.

If your pipeline is designed to ignore events during outages, you should determine each pipeline component mean recovery time to understand the data volume that will be lost during a typical failure. The maximum allowable data loss is a fundamental consideration when designing an at-most-once delivery pipeline.

Most of the pipelines are shared infrastructure. The pipeline is a platform supporting many applications. You should consider whether all current and expected future applications can detect and support data loss due to at-most-once delivery.

You should not assume that during an outage data is always discarded by upstream systems. When recovering from a failure, many systems (especially queues and checkpoint subscribers) read points and resume event transmission from that checkpoint; this is at-least-once delivery.

It is not correct to assume that at-most-once delivery is the default strategy if another is not explicitly chosen. Designing at-most-once delivery requires explicit choices and implementation.

Pattern 7: At-Least-Once Delivery

At-least-once delivery replays recent events starting from an acknowledged (known processed) event. This approach presents some data more than once to the processing pipeline. The typical implementation returns at-least-once delivery checkpoints to a safe point (so you know that they have been processed).

After a failure, the processing is resumed from the checkpoint. As it is possible that events were successfully processed after the checkpoint, these events will be replayed during recovery. This replay means that downstream components will see each event at least once.

There are a number of considerations when using at-least-once delivery:

- This delivery can lead to an unordered event delivery. Note that regardless of the failure model chosen, you should assume that some events will arrive late, unordered, or not arrive.
- Data sources are not well coordinated and rarely are events from sources delivered end to end over a single TCP/IP connection or some other order guaranteeing protocol.
- If the processing operations are not idempotent, replaying events could corrupt and change the output. When designing at-least-once delivery, you must identify and classify processes as idempotent or not.
- If processing operations are not deterministic, replaying events will produce different outputs. Common examples of nondeterministic operations include querying the current clock time or invoking a remote service that could be unavailable.
- This delivery requires a durability contract with upstream components. In the case of failure, some upstream component must have a durable record of the event from which to replay. You should clearly identify durability responsibility through the pipeline, and manage and monitor durable components appropriately. Test operational behavior when the disks fail or are full.

Pattern 8: Exactly-Once Delivery

This type of processing is the ideal because each event is processed exactly once. It avoids the difficult side effects and considerations raised by the other two deliveries. You have exposed the strategies to achieve exactly-once semantics using idempotency in combination with at-least-once delivery.

The following are the fundamental aspects of designing distributed recovery schemas:

- Input streams are usually partitioned across multiple processors
- Inputs can fail on a per partition basis
- Events can be recovered using different strategies

Tag Data Identifiers

When dealing with data streams facing a possible failure, processing each datum exactly once is extremely difficult. If the processing system fails, it may not be easy to determine which data was successfully processed and which was not.

The traditional approaches to this problem are complex, because they require strongly consistent processing systems and smart clients to determine thorough introspection of which data was processed and which was not.

The strongly consistent systems have become scarcer and throughput needs have skyrocketed. This approach has become hard and impractical. Many have failed on precise answers and opted for answers that are as correct as possible under certain circumstances.

As you saw, the Lambda Architecture proposes doing all calculations twice, in two different ways, to provide cross-checks. CRDTs have been proposed as a way to add data structures that can be reasoned when using eventually consistent data stores.

These are not the best options, but idempotency offers another path.

By definition, an *idempotent* operation is an operation that has the same effect no matter how many times it is applied. The simplest example is setting a value. If you set $x = 3.1416$, and then you set $x = 3.1416$ again, the second action doesn't have any effect.

The relation with exactly-once processing is as follows: for idempotent operations, there is no effective difference between at-least-once processing and exactly-once processing. And you know that at-least-once processing is easier to achieve.

One of the core tools used to build robust applications on these platforms is leveraging the idempotent setting of values in eventually consistent systems. Setting individual values is a weaker tool than the twentieth-century ACID transactional model. CRDTs offer more, but come with rigid constraints and restrictions. It is very dangerous to build something without a deep understanding of the offer and how it works.

The advent of consistent systems that truly scale gives a broader set of supported idempotent processing, which can improve and simplify past approaches dramatically. ACID transactions can be built to read and write multiple values based on business logic, while offering the same effects if repeatedly executed.

Pattern 9: Use Upserts over Inserts

An upsert is shorthand for describing a conditional insert. In a nutshell, if the row exists, don't insert it. If the row doesn't exist, insert it.

Some SQL systems have specific syntax for upserts, an ON CONFLICT clause, a MERGE statement, or even a straightforward UPSERT statement. Some NoSQL systems have ways to express the same thing. For key-value stores, the default PUT behavior is an upsert.

When dealing with rows that can be uniquely identified, through a unique key or a unique value, upsert is an idempotent operation.

When the status of an upsert is unclear, often due to the client server or network failure, you can see that it's safe to send it repeatedly until its success can be verified. Note that this type of retry often takes a lot of time to reverse.

Pattern 10: Tag Data with Unique Identifiers

The idempotent operations are difficult when data is not uniquely identifiable. For example, imagine a digital ad application that tracks clicks on a web page. Let's say that an event arrives as a three-value tuple that says user U clicked on spot S at time T with a resolution in seconds. The upsert pattern simply can't be used because it would be possible to record multiple clicks by the same user in the same spot in the same second. This leads to the first subpattern.

Subpattern: Fine-Grained Timestamps

One solution to this click problem is to increase the timestamp resolution to a point at which clicks are unique. If the timestamp allows milliseconds, it is reasonable to assume that the user couldn't click faster than once per millisecond. This enables upsert and idempotency.

It is always critical to verify on the client side that generated events are in fact unique. Trusting a computer's time API to reflect real-world time is a common dangerous mistake. There is a lot of hardware and software that offer milliseconds values, but just on 100ms resolutions. Moreover, the NTP (network time protocol) is able to move clocks backward in many default configurations. Virtualization software is a common example for messing with guest operating system clocks.

To implement it well, you must always check the client side to make sure that the last event and new event have different timestamps before sending them to the server.

Subpattern: Unique IDs at the Event Source

If you can generate a unique id at the client, send that value with the event to ensure that it is unique. If events are generated in one place, it is possible that a simple incrementing counter can uniquely identify events. The trick with a counter is to ensure that you do not use values again after restarting some service.

The following are unique ids approaches:

- Use a central server distributing block of unique ids. A database with strong consistency (e.g., Apache Cassandra) or an agreement system such as ZooKeeper (as you saw on Apache Kafka) can be used to assign blocks. If the event producer fails, then some ids are wasted; 64 bits are enough ids to cover any loss.
- Combine timestamps with ids for uniqueness. If you use millisecond timestamps but want to ensure uniqueness, you start an every-millisecond counter. If two events share a millisecond, give one counter the value 0 and another counter the value 1. This ensures uniqueness.
- Combine timestamps and counters in a 64-bit number. Some databases generate unique ids, dividing 64-bit integers into sections, using 41 bits to identify a millisecond timestamp, 10 bits as a millisecond counter, and 10 bits as an event source id. This leaves one bit for the sign, avoiding issues mixing signed and unsigned integers. Note that 41 bits of milliseconds is about 70 years. Obviously, you can play with the bit sizes for each field, but be careful to anticipate the case where time moves backward.

In case you need something simpler than getting incrementing ids correct, try a universally unique identifier (UUID) library to generate universally unique ids. These work in different ways, but often combine machine information, such as a MAC address, with random values and timestamp values, similar to what was described earlier. The upside is that you can safely assume that UUIDs are unique; the downside is that they often require 16 or more bytes to store.

Pattern 11: Use Kafka Offsets as Unique Identifiers

As you already saw, Apache Kafka has built-in unique identifiers. Combining the topic id with the offset in the log can uniquely identify the event. This sounds like a panacea, but there are reasons to be careful:

- Inserting items into Kafka has the same problems as any other distributed system. Managing exactly-once insertion into Apache Kafka is not easy and it doesn't offer the right tools (at the time of this writing) to manage idempotency when writing to a topic.
- If the Kafka cluster is restarted, topic offsets may no longer be unique. It may be possible to use a third value (e.g., a Kafka cluster id) to make the event unique.

When to Avoid Idempotency

Idempotency can add storage overhead as it stores extra ids for uniqueness. It can add complexity, depending on many factors, such as whether your event processor has certain features or whether your app requires complex operations to be idempotent.

You must evaluate the effort to build an idempotent application against the cost of having imperfect data once in a while. Keep in mind that some data has less value than other data, and spending developer time ensuring that it is perfectly processed may be poor project management.

Another reason to avoid idempotent operations is that the event processor or data store makes it very hard to achieve.

Example: Switch Processing

Consider a phone switch support calls with two events:

1. A request is put on hold (or starts a request).
2. A request is attended.

The system must ingest these events and compute the average hold time.

Case 1: Ordered Requests

In this case, events for a given customer always arrive in the order in which they are generated. Event processing in this example is idempotent, so a request may arrive multiple times, but you can always assume that events arrive in the order they happened.

State schema contains a single tuple containing the total hold time and the total number of hold occurrences. It also contains a set of ongoing holds.

When a request is put on hold (or a request is started), upsert a record into the set of ongoing holds. You use one of the methods described earlier to assign a unique id. Using an upsert instead of an insert makes this operation idempotent.

When a request is attended, you look up the corresponding ongoing hold in the state table. If the ongoing hold is found, you remove it, calculate the duration based on the two correlated events, and update the global hold time and global hold counts accordingly. If this message is seen repeatedly, the ongoing hold record will not be found; the second time it will be processed and can be ignored at that point.

This works quite well, is simple to understand, and is efficient. Guaranteeing order is certainly possible, but it is backstage work. Often it's easier to break the assumption on the processing end, where you may have an ACID consistent processor that makes dealing with complexity easier.

Case 2: Unordered Requests

In this case, requests may arrive in any order. The problem with unordered requests is that you cannot delete from the outstanding holds table when you get a match. What you can do, in a strongly consistent system, is keep one row per hold and mark it as matched when its duration is added to the global duration sum. The row must be kept around to catch repeated messages.

You must hold the events until you are sure that another event for a particular hold could not arrive. This may be minutes, hours, or days, depending on the situation.

This approach is also simple, but requires an additional state. The cost of maintaining an additional state should be weighed against the value of perfectly correct data. It is also possible to drop event records early and allow data to be slightly wrong, but only when events are delayed abnormally. This is a decent compromise between space and correctness in some scenarios.

Summary

In this chapter …

- You read a fast data cookbook.
- You reviewed the fast data concept and compared it with big data and traditional data models.
- You also saw the differences between ACID properties and the CAP theorem. You saw the differences in the term *consistency* in both concepts.
- You reviewed how to integrate modern streaming analytics techniques with transactions.
- You saw how to make streaming transformations to achieve data pipelines processing.
- You learned about fault recovery strategies: at-most-once, at-least-once and exactly-once.
- You learned how to use tag data identifiers.

In the next chapter, you apply these concepts to data pipeline patterns.