O'REILLY®

Fast Data: Smart and at Scale

Design Patterns and Recipes

Ryan Betts & John Hugg

SAN JOSE

LONDON

NEW YORK

SINGAPORE

Strata+ Hadoop WORLD

Make Data Work

strataconf.com

Presented by O'Reilly and Cloudera, Strata + Hadoop World is where cutting-edge data science and new business fundamentals intersect—and merge.

- Learn business applications of data technologies
- Develop new skills through trainings and in-depth tutorials
- Connect with an international community of thousands who work with data

Job # 15420

Fast Data: Smart and at Scale

Design Patterns and Recipes

Ryan Betts and John Hugg

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Fast Data: Smart and at Scale
by Ryan Betts and John Hugg

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editor: Tim McGovern
Production Editor: Dan Fauxsmith

Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

September 2015: First Edition

Revision History for the First Edition
2015-09-01: First Release
2015-10-20: Second Release

978-1-491-94038-9

[LSI]

Table of Contents

Foreword

We are witnessing tremendous growth of the scale and rate at which data is generated. In earlier days, data was primarily generated as a result of a real-world human action—the purchase of a product, a click on a website, or the pressing of a button. As computers become increasingly independent of humans, they have started to generate data at the rate at which the CPU can process it—a furious pace that far exceeds human limitations. Computers now initiate trades of stocks, bid in ad auctions, and send network messages completely independent of human involvement.

This has led to a reinvigoration of the data-management community, where a flurry of innovative research papers and commercial solutions have emerged to address the challenges born from the rapid increase in data generation. Much of this work focuses on the problem of collecting the data and analyzing it in a period of time after it has been generated. However, an increasingly important alternative to this line of work involves building systems that process and analyze data immediately after it is generated, feeding decision-making software (and human decision makers) with actionable information at low latency. These "fast data" systems usually incorporate recent research in the areas of low-latency data stream management systems and high-throughput main-memory database systems.

As we become increasingly intolerant of latency from the systems that people interact with, the importance and prominence of fast data will only grow in the years ahead.

—Daniel Abadi, Ph.D.
Associate Professor, Yale University

Fast Data Application Value

Looking Beyond Streaming

Fast data application deployments are exploding, driven by the Internet of Things (IoT), a surge in data from machine-to-machine communications (M2M), mobile device proliferation, and the revenue potential of acting on fast streams of data to personalize offers, interact with customers, and automate reactions and responses.

Fast data applications are characterized by the need to ingest vast amounts of streaming data; application and business requirements to perform analytics in real time; and the need to combine the output of real-time analytics results with transactions on live data. Fast data applications are used to solve three broad sets of challenges: streaming analytics, fast data pipeline applications, and request/response applications that focus on interactions.

While there's recognition that fast data applications produce significant value—fundamentally different value from big data applications—it's not yet clear which technologies and approaches should be used to best extract value from fast streams of data.

Legacy relational databases are overwhelmed by fast data's requirements, and existing tooling makes building fast data applications challenging. NoSQL solutions offer speed and scale but lack transactionality and query/analytics capability. Developers sometimes stitch together a collection of open source projects to manage the data stream; however, this approach has a steep learning curve, adds complexity, forces duplication of effort with hybrid batch/streaming approaches, and limits performance while increasing latency.

So how do you combine real-time, streaming analytics with real-time decisions in an architecture that's reliable, scalable, and simple? You could do it yourself using a batch/streaming approach that would require a lot of infrastructure and effort; or you could build your app on a fast, distributed data processing platform with support for per-event transactions, streaming aggregations combined with per-event ACID processing, and SQL. This approach would simplify app development and enhance performance and capability.

This report examines how to develop apps for fast data, using well-recognized, predefined patterns. While our expertise is with VoltDB's unified fast data platform, these patterns are general enough to suit both the do-it-yourself, hybrid batch/streaming approach as well as the simpler, in-memory approach.

Our goal is to create a collection of "fast data app development recipes." In that spirit, we welcome your contributions, which will be tested and included in future editions of this report. To submit a recipe, send a note to *recipes@fastsmartatscale.com*.

Fast Data and the Enterprise

The world is becoming more interactive. Delivering information, offers, directions, and personalization to the right person, on the right device, at the right time and place—all are examples of new fast data applications. However, building applications that enable real-time interactions poses a new and unfamiliar set of data-processing challenges. This report discusses common patterns found in fast data applications that combine streaming analytics with operational workloads.

Understanding the structure, data flow, and data management requirements implicit in these fast data applications provides a foundation to evaluate solutions for new projects. Knowing some common patterns (recipes) to overcome expected technical hurdles makes developing new applications more predictable—and results in applications that are more reliable, simpler, and extensible.

New fast data application styles are being created by developers working in the cloud, IoT, and M2M. These applications present unfamiliar challenges. Many of these applications exceed the scale of traditional tools and techniques, creating new challenges not solved by traditional legacy databases that are too slow and don't scale out. Additionally, modern applications scale across multiple machines, connecting multiple systems into coordinated wholes, adding complexity for application developers.

As a result, developers are reaching for new tools, new design techniques, and often are tasked with building distributed systems that require different thinking and different skills than those gained from past experience.

This report is structured into four main sections: an introduction to fast data, with advice on identifying and structuring fast data architectures; a chapter on ACID and CAP, describing why it's important to understand the concepts and limitations of both in a fast data architecture; four chapters, each a recipe/design pattern for writing certain types of streaming/fast data applications; and a glossary of terms and concepts that will aid in understanding these patterns. The recipe portion of the book is designed to be easily extensible as new common fast data patterns emerge. We invite readers to submit additional recipes at *recipes@fastsmartatscale.com*

CHAPTER 1
What Is Fast Data?

Into a world dominated by discussions of big data, fast data has been born with little fanfare. Yet fast data will be the agent of change in the information-management industry, as we will show in this report.

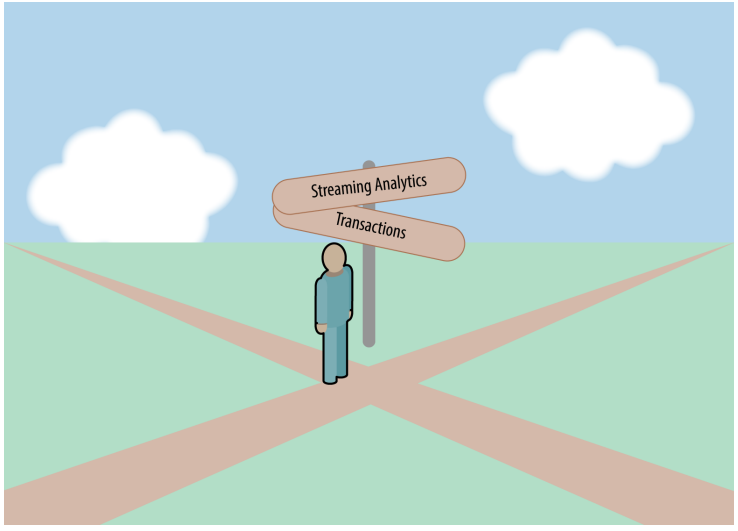
Fast data is data in motion, streaming into applications and computing environments from hundreds of thousands to millions of endpoints—mobile devices, sensor networks, financial transactions, stock tick feeds, logs, retail systems, telco call routing and authorization systems, and more. Real-time applications built on top of fast data are changing the game for businesses that are data dependent: telco, financial services, health/medical, energy, and others. It's also changing the game for developers, who must build applications to handle increasing streams of data.[1]

We're all familiar with big data. It's data at rest: collections of structured and unstructured data, stored in Hadoop and other "data lakes," awaiting historical analysis. Fast data, by contrast, is stream-

1 Where is all this data coming from? We've all heard the statement that "data is doubling every two years"—the so-called Moore's Law of data. And according to the oft-cited EMC Digital Universe Study (2014), which included research and analysis by IDC, this statement is true. The study states that data "will multiply 10-fold between 2013 and 2020—from 4.4 trillion gigabytes to 44 trillion gigabytes". This data, much of it new, is coming from an increasing number of new sources: people, social, mobile, devices, and sensors. It's transforming the business landscape, creating a generational shift in how data is used, and a corresponding market opportunity. Applications and services tapping this market opportunity require the ability to process data fast.

ing data: data in motion. Fast data demands to be dealt with as it streams in to the enterprise *in real time*. Big data can be dealt with some other time—typically after it's been stored in a Hadoop data warehouse—and analyzed *via batch processing*.

A stack is emerging across verticals and industries to help developers build applications to process fast streams of data. This fast data stack has a unique purpose: to process real-time data and output recommendations, analytics, and decisions—transactions—in milliseconds (billing authorization and up-sell of service level, for example, in telecoms), although some fast data use cases can tolerate up to minutes of latency (energy sensor networks, for example).



Applications of Fast Data

Fast data applications share a number of requirements that influence architectural choices. Three of particular interest are:

- Rapid *ingestion* of millions of data events—streams of live data from multiple endpoints
- *Streaming analytics* on incoming data
- *Per-event transactions* made on live streams of data in real time as events arrive.

Ingestion

Ingestion is the first stage in the processing of streaming data. The job of ingestion is to interface with streaming data sources and to accept and transform or normalize incoming data. Ingestion marks the first point at which data can be transacted against, applying key functions and processes to extract value from data—value that includes insight, intelligence, and action.

Developers have two choices for ingestion. The first is to use "direct ingestion," where a code module hooks directly into the data-generating API, capturing the entire stream at the speed at which the API and the network will run, e.g., at "wire speed." In this case, the analytic/decision engines have a direct ingestion "adapter." With some amount of coding, the analytic/decision engines can handle streams of data from an API pipeline without the need to stage or cache any data on disk.

If access to the data-generating API is not available, an alternative is using a message queue, e.g., Kafka. In this case, an ingestion system processes incoming data from the queue. Modern queuing systems handle partitioning, replication, and ordering of data, and can manage backpressure from slower downstream components.

Streaming Analytics

As data is created, it arrives in the enterprise in fast-moving streams. Data in a stream may arrive in many data types and formats. Most often, the data provides information about the process that generated it; this information may be called messages or events. This includes data from new sources, such as sensor data, as well as clickstreams from web servers, machine data, and data from devices, events, transactions, and customer interactions.

The increase in fast data presents the opportunity to perform analytics on data as it streams in, rather than post-facto, after it's been pushed to a data warehouse for longer-term analysis. The ability to analyze streams of data and make in-transaction decisions on this fresh data is the most compelling vision for designers of data-driven applications.

Per-Event Transactions

As analytic platforms mature to produce real-time summary and reporting on incoming data, the speed of analysis exceeds a human operator's ability to act. To derive value from real-time analytics, one must be able to take action in real time. This means being able to transact against event data as it arrives, using real-time analysis in combination with business logic to make optimal decisions—to detect fraud, alert on unusual events, tune operational tolerances, balance work across expensive resources, suggest personalized responses, or tune automated behavior to real-time customer demand.

At a data-management level, all of these actions mean being able to read and write multiple, related pieces of data together, recording results and decisions. It means being able to transact against each event as it arrives.

High-speed streams of incoming data can add up to massive amounts of data, requiring systems that ensure high availability and at-least-once delivery of events. It is a significant challenge for enterprise developers to create apps not only to ingest and perform analytics on these feeds of data, but also to capture value, via per-event transactions, from them.

Uses of Fast Data

Front End for Hadoop

Building a fast front end for Hadoop is an important use of fast data application development. A fast front end for Hadoop should perform the following functions on fast data: filter, dedupe, aggregate, enrich, and denormalize. Performing these operations on the front end, before data is moved to Hadoop, is much easier to do in a fast data front end than it is to do in batch mode, which is the approach used by Spark Streaming and the Lambda Architecture. Using a fast front end carries almost zero cost in time to do filter, deduce, aggregate, etc., at ingestion, as opposed to doing these operations in a separate batch job or layer. A batch approach would need to clean the data, which would require the data to be stored twice, also introducing latency to the processing of data.

An alternative is to dump everything in HDFS and sort it all out later. This is easy to do at ingestion time, but it's a big job to sort out later. Filtering at ingestion time also eliminates bad data, data that is too old, and data that is missing values; developers can fill in the values, or remove the data if it doesn't make sense.

Then there's aggregation and counting. Some developers maintain it's difficult to count data at scale, but with an ingestion engine as the fast front end of Hadoop it's possible to do a tremendous amount of counting and aggregation. If you've got a raw stream of data, say 100,000 events per second, developers can filter that data by several orders of magnitude, using counting and aggregations, to produce less data. Counting and aggregations reduce large streams of data and make it manageable to stream data into Hadoop.

Developers also can delay sending aggregates to HDFS to allow for late-arriving events in windows. This is a common problem with other streaming systems—data streams in a few seconds too late to a window that has already been sent to HDFS. A fast data front end allows developers to update aggregates when they come in.

Enriching Streaming Data

Enrichment is another option for a fast data front end for Hadoop.

Streaming data often needs to be filtered, correlated, or enriched before it can be "frozen" in the historical warehouse. Performing this processing in a streaming fashion against the incoming data feed offers several benefits:

1. Unnecessary latency created by batch ETL processes is eliminated and time-to-analytics is minimized.
2. Unnecessary disk IO is eliminated from downstream big data systems (which are usually disk-based, not memory-based, when ETL is real time and not batch oriented).
3. Application-appropriate data reduction at the ingest point eliminates operational expense downstream—less hardware is necessary.

The input data feed in fast data applications is a stream of information. Maintaining stream semantics while processing the events in the stream discretely creates a clean, composable processing model. Accomplishing this requires the ability to act on each input event—a

capability distinct from building and processing windows, as is done in traditional CEP systems.

These per-event actions need three capabilities: fast look-ups to enrich each event with metadata; contextual filtering and sessionizing (re-assembly of discrete events into meaningful logical events is very common); and a stream-oriented connection to downstream pipeline systems (e.g., distributed queues like Kafka, OLAP storage, or Hadoop/HDFS clusters). This requires a stateful system fast enough to transact on a per-event basis against unlimited input streams and able to connect the results of that transaction processing to downstream components.

Queryable Cache

Queries that make a decision on ingest are another example of using fast data front-ends to deliver business value. For example, a click event arrives in an ad-serving system, and we need to know which ad was shown, and analyze the response to the ad. Was the click fraudulent? Was it a robot? Which customer account do we debit because the click came in and it turns out that it wasn't fraudulent? Using queries that look for certain conditions, we might ask questions such as: "Is this router under attack based on what I know from the last hour?" Another example might deal with SLAs: "Is my SLA being met based on what I know from the last day or two? If so, what is the contractual cost?" In this case, we could populate a dashboard that says SLAs are not being met, and it has cost *n* in the last week. Other deep analytical queries, such as "How many purple hats were sold on Tuesdays in 2015 when it rained?" are really best served by systems such as Hive or Impala. These types of queries are ad-hoc and may involve scanning lots of data; they're typically not fast data queries.

CHAPTER 2

Disambiguating ACID and CAP

Fast data is transformative. The most significant uses for fast data apps have been discussed in prior chapters. Key to writing fast data apps is an understanding of two concepts central to modern data management: the ACID properties and the CAP theorem, addressed in this chapter. It's unfortunate that in both acronyms the "C" stands for "Consistency," but actually means completely different things. What follows is a primer on the two concepts and an explanation of the differences between the two "C"s.

What Is ACID?

The idea of transactions, their semantics and guarantees, evolved with data management itself. As computers became more powerful, they were tasked with managing more data. Eventually, multiple users would share data on a machine. This led to problems where data could be changed or overwritten out from under users in the middle of a calculation. Something needed to be done; so the academics were called in.

The rules were originally defined by Jim Gray in the 1970s, and the acronym was popularized in the 1980s. "ACID" transactions solve many problems when implemented to the letter, but have been engaged in a push-pull with performance tradeoffs ever since. Still, simply understanding these rules can educate those who seek to bend them.

A transaction is a bundling of one or more operations on database state into a single sequence. Databases that offer transactional semantics offer a clear way to start, stop, and cancel (or roll back) a set of operations (reads and writes) as a single logical meta-operation.

But transactional semantics do not make a "transaction." A true transaction must adhere to the ACID properties. ACID transactions offer guarantees that absolve the end user of much of the headache of concurrent access to mutable database state.

From the seminal Google F1 Paper:

> The system must provide ACID transactions, and must always present applications with consistent and correct data. Designing applications to cope with concurrency anomalies in their data is very error-prone, time-consuming, and ultimately not worth the performance gains.

What Does ACID Stand For?

- **Atomic**: All components of a transaction are treated as a single action. All are completed or none are; if one part of a transaction fails, the database's state is unchanged.
- **Consistent**: Transactions must follow the defined rules and restrictions of the database, e.g., constraints, cascades, and triggers. Thus, any data written to the database must be valid, and any transaction that completes will change the state of the database. No transaction will create an invalid data state. Note this is different from "consistency" as defined in the CAP theorem.
- **Isolated**: Fundamental to achieving concurrency control, isolation ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially, i.e., one after the other; with isolation, an incomplete transaction cannot affect another incomplete transaction.
- **Durable**: Once a transaction is committed, it will persist and will not be undone to accommodate conflicts with other operations. Many argue that this implies the transaction is on disk as well; most formal definitions aren't specific.

What Is CAP?

CAP is a tool to explain tradeoffs in distributed systems. It was presented as a conjecture by Eric Brewer at the 2000 Symposium on Principles of Distributed Computing, and formalized and proven by Gilbert and Lynch in 2002.

What Does CAP Stand For?

- **Consistent**: All replicas of the same data will be the same value across a distributed system.
- **Available**: All live nodes in a distributed system can process operations and respond to queries.
- **Partition Tolerant**: The system will continue to operate in the face of arbitrary network partitions.

The most useful way to think about CAP:

> In the face of network partitions, you can't have both perfect consistency and 100% availability. Plan accordingly.

To be clear, CAP isn't about what is possible, but rather, what isn't possible. Thinking of CAP as a "You-Pick-Two" theorem is misguided and dangerous. First, "picking" AP or CP doesn't mean you're actually going to be perfectly consistent or perfectly available; many systems are neither. It simply means the designers of a system have at some point in their implementation favored consistency or availability when it wasn't possible to have both.

Second, of the three pairs, CA isn't a meaningful choice. The designer of distributed systems does not simply make a decision to ignore partitions. The potential to have partitions is one of the definitions of a distributed system. If you don't have partitions, then you don't have a distributed system, and CAP is just not interesting. If you do have partitions, ignoring them automatically forfeits C, A, or both, depending on whether your system corrupts data or crashes on an unexpected partition.

How Is CAP Consistency Different from ACID Consistency?

ACID consistency is all about database rules. If a schema declares that a value must be unique, then a consistent system will enforce uniqueness of that value across all operations. If a foreign key implies deleting one row will delete related rows, then a consistent system will ensure the state can't contain related rows once the base row is deleted.

CAP consistency promises that every replica of the same logical value, spread across nodes in a distributed system, has the same exact value at all times. Note that this is a logical guarantee, rather than a physical one. Due to the speed of light, it may take some non-zero time to replicate values across a cluster. The cluster can still present a logical view of preventing clients from viewing different values at different nodes.

The most interesting confluence of these concepts occurs when systems offer more than a simple key-value store. When systems offer some or all of the ACID properties across a cluster, CAP consistency becomes more involved. If a system offers repeatable reads, compare-and-set or full transactions, then to be CAP consistent, it must offer those guarantees at any node. This is why systems that focus on CAP availability over CAP consistency rarely promise these features.

What Does "Eventual Consistency" Mean in This Context?

Let's consider the simplest case, a two-server cluster. As long as there are no failures, writes are propagated to both machines and everything hums along. Now imagine the network between nodes is cut. Any write to a node now will not propagate to the other node. State has diverged. Identical queries to the two nodes may give different answers.

The traditional response is to write a complex rectification process that, when the network is fixed, examines both servers and tries to repair and resynchronize state.

"Eventual Consistency" is a bit overloaded, but aims to address this problem with less work for the developer. The original Dynamo

paper formally defined EC as the method by which multiple replicas of the same value may differ temporarily, but would eventually converge to a single value. This guarantee that divergent data would be temporary can render a complex repair and resync process unnecessary.

EC doesn't address the issue that state still diverges temporarily, allowing answers to queries to differ based on where they are sent. Furthermore, EC doesn't promise that data will converge to the newest or the most correct value (however that is defined), merely that it will converge.

Numerous techniques have been developed to make development easier under these conditions, the most notable being Conflict-free Replicated Data Types (CRDTs), but in the best cases, these systems offer fewer guarantees about state than CAP-consistent systems can. The benefit is that under certain partitioned conditions, they may remain available for operations in some capacity.

It's also important to note that Dynamo-style EC is very different from the log-based rectification used by the financial industry to move money between accounts. Both systems are capable of diverging for a period of time, but the bank's system must do more than eventually agree; banks have to eventually have the right answer.

The next chapters provide examples of how to conceptualize and write fast data apps.

CHAPTER 3

Recipe: Integrate Streaming Aggregations and Transactions

Idea in Brief

Increasing numbers of high-speed transactional applications are being built: operational applications that transact against a stream of incoming events for use cases like real-time authorization, billing, usage, operational tuning, and intelligent alerting. Writing these applications requires combining real-time analytics with transaction processing.

Transactions in these applications require real-time analytics as inputs. Recalculating analytics from base data for each event in a high-velocity feed is impractical. To scale, maintain streaming aggregations that can be read cheaply in the transaction path. Unlike periodic batch operations, streaming aggregations maintain consistent, up-to-date, and accurate analytics needed in the transaction path.

This pattern trades ad hoc analytics capability for high-speed access to analytic outputs that are known to be needed by an application. This trade-off is necessary when calculating an analytic result from base data for each transaction is infeasible.

Let's consider a few example applications to illustrate the concept.

Pattern: Reject Requests Past a Threshold

Consider a high-request-volume API that must implement sophisticated usage metrics for groups of users and individual users on a per-operation basis. Metrics are used for multiple purposes: they are used to derive usage-based billing charges, and they are used to enforce a contracted quality of service standard (expressed as a number of requests per second, per user, and per group). In this case, the operational platform implementing the policy check must be able to maintain fast counters for API operations, for users and for groups. These counters must be accurate (they are inputs to billing and quality of service policy enforcement), and they must be accessible in real time to evaluate and authorize (or deny) new requests.

In this scenario, it is necessary to keep a real-time balance for each user. Maintaining the balance accurately (granting new credits, deducting used credits) requires an ACID OLTP system. That same system requires the ability to maintain high-speed aggregations. Combining real-time, high-velocity streaming aggregations with transactions provides a scalable solution.

Pattern: Alerting on Variations from Predicted Trends

Imagine an operational monitoring platform that needs to issue alerts or alarms when a threshold exceeds the predicated trend line to a statistically significant level. This system combines two capabilities: it must maintain real-time analytics (streaming aggregations, counters, and summary state of the current utilization), and it must be able to compare these to the predicated trend. If the trend is exceeded, the system must generate an alert or alarm. Likely, the system will record this alarm to suppress an alarm storm (to throttle the rate of alarm publishing for a singular event).

This is another system that requires the combination of analytical and transactional capability. Without the combined capability, this problem would need three separate systems working in unison: an analytics system that is micro-batching real-time analytics; an application reading those analytics and reading the predicated trendline to generate alerts and alarms; and a transactional system that is storing generated alert and alarm data to implement the suppression

logic. Running three tightly coupled systems like this (the solution requires all three systems to be running) lowers reliability and complicates operations.

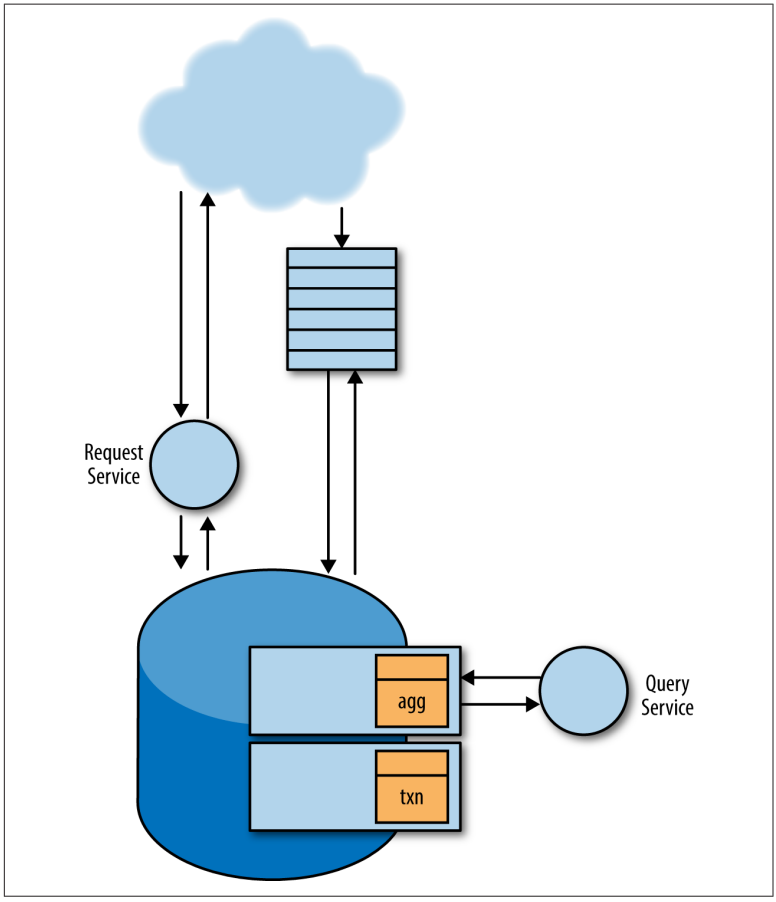


Figure 3-1. Streaming aggregations with transactions

Combining streaming event processing with request-response style applications allows operationalizing real-time analytics.

When to Avoid This Pattern

Traditional OLAP systems offer the benefit of fast analytic queries without pre-aggregation. These systems can execute complex queries that scan vast amounts of data in seconds to minutes—insufficient for high-velocity event feeds but within the threshold for many

batch reporting functions, data science, data exploration, and human analyst workflows. However, these systems do not support high-velocity transactional workloads. They are optimized for reporting, not for OLTP-style applications.

Related Concepts

Pre-aggregation is a common technique for which many algorithms and features have been developed. Materialized views, probabilistic data structures (examples: HyperLogLog, Bloom filters), and windowing are common techniques for implementing efficient real-time aggregation and summary state.

1. Materialized Views: a view defines an aggregation, partitioning, filter or join, grouping of base data. "Materialized" views maintain a physical copy of the resulting tuples. Materialized Views allow declarative aggregations, eliminating user code and enabling succinct, correct, and easy aggregations.
2. Probabilistic data structures aggregate data within some probabilistically bounded margin of error. These algorithms typically trade precision for space, enabling bounded estimation in a much smaller storage footprint. Examples of probabilistic data structures include Bloom filters and HyperLogLog algorithms.
3. Windows are used to express moving averages, or time-windowed summaries of a continuous event timeline. These techniques are often found in CEP style or micro-batching systems. SQL analytic functions (OVER, PARTITION) bring this functionality to SQL platforms.

CHAPTER 4

Recipe: Design Data Pipelines

Idea in Brief

Processing big data effectively often requires multiple database engines, each specialized to a purpose. Databases that are very good at event-oriented real-time processing are likely not good at batch analytics against large volumes. Some systems are good for high-velocity problems. Others are good for large-volume problems. However, in most cases, these systems need to interoperate to support meaningful applications.

Minimally, data arriving at the high-velocity, ingest-oriented systems needs to be processed and captured into the volume-oriented systems. In more advanced cases, reports, analytics, and predictive models generated from the volume-oriented systems need to be communicated to the velocity-oriented system to support real-time applications. Real-time analytics from the velocity side need to be integrated into operational dashboards or downstream applications that process real-time alerts, alarms, insights, and trends.

In practice, this means that many big data applications sit on top of a platform of tools. Usually the components of the platform include at least a large shared storage pool (like HDFS), a high-performance BI analytics query tool (like a columnar SQL system), a batch processing system (MapReduce or perhaps Spark), and a streaming system. Data and processing outputs move between all of these systems. Designing that dataflow—designing a processing pipeline

—that coordinates these different platform components, is key to solving many big data challenges.

Pattern: Use Streaming Transformations to Avoid ETL

New events being captured into a long-term repository often require transformation, filtering, or processing before they are available for reporting use cases. For example, many applications capture sessions comprising several discrete events, enrich events with static dimension data to avoid expensive repeated joins in the batch layer, or filter redundant events from a dataset storing only unique values.

There are at least two approaches to running these transformations.

1. All of the data can be landed to a long-term repository and then extracted, transformed, and re-loaded back in its final form. This approach trades I/O, additional storage requirements, and longer time to insight (reporting is delayed until the ETL process completes) for a slightly simpler architecture (likely data is moving directly from a queue to HDFS). This approach is sometimes referred to as schema-on-read. It narrows the choice of backend systems to those systems that are relatively schema-free—systems that may not be optimal, depending on your specific reporting requirements.
2. The transformations can be executed in a streaming fashion before the data reaches the long-term repository. This approach adds a streaming component between the source queue and the final repository, creating a continuous processing pipeline. Moving the transformation to a real-time processing pipeline has several advantages. The write I/O to the backend system is at least halved (in the first model, first raw data is written and then ETL'd data is written. In this model, only ETL'd data is written.) This leaves more I/O budget available for data science and reporting activity—the primary purpose of the backend repository. Operational errors are noticeable in almost real time. When using ETL, raw data is not inspected until the ETL process runs. This delays operational notifications of missing or corrupt inputs. Finally, time-to-insight is reduced. For example, when organizing session data, a session is available to participate in batch reporting immediately upon being "closed." When

ETL'ing, you must wait, on average, for half of the ETL period before the data is available for backend processing.

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

Real-time applications processing incoming events often require analytics from backend systems. For example, if writing an alerting system that issues notifications when a moving five minute interval exceeds historical patterns, the data describing the historical pattern needs to be available. Likewise, applications managing real-time customer experience or personalization often use customer segmentation reports generated by statistical analysis run on the batch analytics system. A third common example is hosting OLAP outputs in a fast, scalable query cache to support operators and applications that need high-speed, highly concurrent access to data.

In many cases, the reports, analytics, or models from big data analytics need to be made available to real-time applications. In this case, information flows from the big data (batch) oriented system to the high-velocity (streaming) system. This introduces a few important requirements. First, the fast data, velocity-oriented application requires a data management system capable of holding the state generated by the batch system; second, this state needs to be regularly updated or replaced in full. There are a few common ways to manage the refresh cycle—the best tradeoff will depend on your specific application.

Some applications (for example, 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 basis is sufficient. Updates will need to communicate new records (in this example, new customers), updates to existing records (customers that have been recategorized), and deletions (ex-customers). Records in the velocity system should be timestamped for operational monitoring and alerts generated if stale records persist beyond the expected refresh cycle.

Other applications require the analytics data to be strictly consistent; if it is insufficient for each record to be internally consistent, the set of records as a whole requires a consistency guarantee. These cases are often seen in analytic query caches. Often these caches are quer-

ied 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 these report data from the batch analytics system to the real-time system is to write the data to a shadow table. Once the shadow table is completely written, it can be atomically renamed, or swapped, with the main table that is 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 will never see a mix of data from both reports in a single query.

Pattern: Use Loose Coupling to Improve Reliability

When connecting multiple systems, it is imperative that all systems have an independent fate. Any part of the pipeline should be able to fail while leaving other systems available and functional. If the batch backend is offline, the high-velocity front end should still be operating, and vice versa. This requires thinking through several design decisions:

1. Where does data that can not be pushed (or pulled) through the pipeline rest? Which components are responsible for the durability of stalled data?
2. How do systems recover? Which systems are systems of record —meaning they can be recovery sources for lost data or interrupted processing?
3. What is the failure and availability model of each component in the pipeline?
4. When a component fails, which other components become unavailable? How long can upstream components maintain functionality (for example, how long can they log processed work to disk)? These numbers inform your recovery time objective (RTO).

In every pipeline, there is by definition a slowest component—a bottleneck. When designing, explicitly choose the component that will be your bottleneck. Having many systems, each with identical performance, means a minor degradation to any system will create a new overall bottleneck. This is operationally painful. It is often bet-

ter to choose your most reliable component as your bottleneck or your most expensive resource as your bottleneck. Overall you will achieve a more predictable level of reliability.

When to Avoid Pipelines

Tying multiple systems together is always complex. This complexity is rarely linear with the number of systems. Typically, complexity increases as a function of connections (and worst case you can have N^2 connections between N systems). To the extent your problem can be fully solved by a single stand-alone system, you should prefer that approach. However, large-volume and high-velocity data management problems typically require a combination of specialized systems. When combining these systems, carefully, consciously design dataflow and connections between them. Loosely couple systems so each operates independently of the failure of its connected partners. Use multiple systems to simplify when possible—for example, by moving batch ETL processes to continuous processes.

CHAPTER 5

Recipe: Pick Failure-Recovery Strategies

Idea in Brief

Most streaming applications move data through multiple processing stages. In many cases, events are landed in a queue and then read by downstream components. Those components might write new data back to a queue as they process or they might directly stream data to their downstream components. Building a reliable data pipeline requires designing failure-recovery strategies.

With multiple processing stages connected, eventually one stage will fail, become unreachable, or otherwise unavailable. When this occurs, the other stages continue to receive data. When the failed component comes back online, typically it must recover some previous state and then begin processing new events. This recipe discusses where to resume processing.

The idempotency recipe discusses a specific technique to achieve exactly-once semantics.

Additionally, for processing pipelines that are horizontally scaled, where each stage has multiple servers or process running in parallel, a subset of servers within the cluster can fail. In this case, the failed server needs to be recovered, or its work needs to be reassigned.

There are a few factors that complicate these problems and lead to different trade-offs.

First, it is usually uncertain what the last processed event was. It is typically not technologically feasible, for example, to two-phase commit the event processing across all pipeline components. Typically, unreliable communication between processing components means the fate of events in-flight near the time of failure is unknown.

Second, event streams are often partitioned (sharded) across a number of processors. Processors and upstream sources can fail in arbitrary combinations. Picturing a single, unified event flow is often an insufficient abstraction. Your system should assume that a single partition of events can be omitted from an otherwise available stream due to failure.

This leads to three options for resuming processing distinguished by how events near the failure time are handled. Approaches to solving the problem follow.

Pattern: At-Most-Once Delivery

At-most-once delivery allows some events to be dropped. In these cases, events not processed because of an interruption are simply dropped. They do not become part of the input to the downstream system. If the data itself is low value or loses value if it is not immediately processed, this may be acceptable.

Questions you should answer when considering at-most-once delivery include:

- Will historical analytics know that data was unavailable?
- Will the event stream eventually be archived to an OLAP store or data lake? If so, how will future reporting and data science algorithms detect and manage the missing values?
- Is it clear what data was lost?

Remember that lost data is unlikely to align exactly with session boundaries, time windows, or even data sources. It is also likely that only partial data was dropped during the outage period—meaning that some values are present.

Additional questions to be answered include: Is there a maximum period of outage? What is the largest gap that can be dropped?

If the pipeline is designed to ignore events during outages, you should determine the mean time to recovery for each component of the pipeline to understand what volume of data will be lost in a typical failure. Understanding the maximum allowable data loss should be an explicit consideration when planning an at-most-once delivery pipeline.

Many data pipelines are shared infrastructure. The pipeline is a platform that supports many applications. You should consider whether all current and all expected future applications can detect and tolerate data loss due to at-most-once delivery.

Finally, it is incorrect to assume that at-most-once delivery is the "default" if another strategy is not explicitly chosen. You should not assume that data during an outage is always discarded by upstream systems. Many systems, queues especially, checkpoint subscriber read points and resume event transmission from the checkpoint when recovering from a failure. (This is actually an example of at-least-once delivery.) Designing at-most-once delivery requires explicit choices and implementation—it is not the "free" choice.

Pattern: At-Least-Once Delivery

At-least-once delivery replays recent events starting from a known-processed (acknowledged) event. This approach presents some data to the processing pipeline more than once. The typical implementation backing at-least-once delivery checkpoints a safe-point (that is known to have been processed). After a failure, processing resumes from the checkpoint. It is likely that events were successfully processed after the checkpoint. These events will be replayed during recovery. This replay means that downstream components see each event "at least once."

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

At-least-once delivery can lead to out-of-order event delivery. In reality, regardless of the failure model chosen, you should assume that some events will arrive late, out of order, or not at all.

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 processing operations are not idempotent, replaying events will corrupt or change outputs. When designing at-least-once delivery, identify and characterize processes as idempotent or not.

If processing operations are not deterministic, replaying events will produce different outcomes. Common examples of non-deterministic operations include querying the current wallclock time or invoking a remote service (that may be unavailable).

At-least-once 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, testing operational behavior when disks fail or fill.

Pattern: Exactly-Once Delivery

Exactly-once processing is the ideal—each event is processed exactly once. This avoids the difficult side effects and considerations raised by at-most-once and at-least-once processing. See Chapter 6 for strategies on achieving exactly-once semantics using idempotency in combination with at-least-once delivery.

Understanding that input streams are typically partitioned across multiple processors, that inputs can fail on a per-partition basis, and that events can be recovered using different strategies are all fundamental aspects of designing distributed recovery schemes.

CHAPTER 6

Recipe: Combine At-Least-Once Delivery with Idempotent Processing to Achieve Exactly-Once Semantics

Idea in Brief

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

Traditional approaches to this problem are complex, require strongly consistent processing systems, and require smart clients that can determine through introspection what has or hasn't been processed.

As strongly consistent systems have become more scarce, and throughput needs have skyrocketed, this approach often has been deemed unwieldy and impractical. Many have given up on precise answers and chosen to work toward answers that are as correct as possible under the circumstances. The Lambda Architecture proposes doing all calculations twice, in two different ways, to allow for cross-checking. Conflict-free replicated data types (CRDTs) have been proposed as a way to add data structures that can be reasoned about when using eventually consistent data stores.

If these options are less than ideal, idempotency offers another path.

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 I set $x = 5$, then I set $x = 5$ again, the second action doesn't have any effect. How does this relate to exactly-once processing? For idempotent operations, there is no effective difference between at-least-once processing and exactly-once processing, and at-least-once processing is much easier to achieve.

Leveraging the idempotent setting of values in eventually consistent systems is one of the core tools used to build robust applications on these platforms. Nevertheless, setting individual values is a much weaker tool than the ACID-transactional model that pushed data management forward in the late 20th century. CRDTs offer more, but come with rigid constraints and restrictions. They're still a dangerous thing to build around without a deep understanding of what they offer and how they work.

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

Pattern: Use Upserts Over Inserts

An upsert is shorthand for describing a conditional insert; if the row exists, don't insert; if the row does not, insert it. Some systems support specific syntax for this. In SQL, this can involve 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 behavior of "put" is an upsert. Check your system's documentation.

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

When the status of an upsert is unclear, often due to client, server node, or network failure, it is safe to send repeatedly until its success can be verified. Note that this type of retry often should make use of exponential backoff.

Pattern: Tag Data with Unique Identifiers

Idempotent operations are more difficult when data isn't uniquely identifiable. Imagine a digital ad-tech app that tracks clicks on a web page. An event arrives as a three-tuple that says user X clicked on spot Y at time T (with second resolution). With this design, the upsert pattern 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, e.g., a double-click.

Subpattern: Fine-Grained Timestamps

One solution to the non-unique clicks problem is to increase the timestamp resolution to a point at which clicks are unique. If the timestamp stored milliseconds, it might be reasonable to assume that a user couldn't click faster than once per millisecond. This makes upsert usable and idempotency attainable.

Note that it's critical to verify on the client side that generated events are in fact unique. Trusting a computer time API to accurately reflect real-world time is a common mistake, proven time and again to be dangerous. For example, some hardware/software/APIs offer millisecond values, but 100 ms resolution. NTP (network time protocol) is able to move clocks backward in many default configurations. Virtualization software is notorious for messing with guest OS clocks.

To do this well, check that the last event and the new event have different times on the client side before sending the event 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 tuple to ensure that it is unique. If events are generated in one place, it's possible that a simple incrementing counter can uniquely identify events. The trick with a counter is to ensure you don't re-use values after restarting some service.

One approach is to use a central server to dole out blocks of unique ids. A database with strong consistency or an agreement system such as ZooKeeper can be used to assign blocks of ten thousand, one hundred thousand, or one million ids in chunks. If the event producer fails, then some ids are wasted, but 64 bits should have enough ids to cover any loss.

Another approach is to combine timestamps with ids for uniqueness. Are you using millisecond timestamps but want to ensure uniqueness? Start a new counter for every millisecond. If two events share a millisecond, give one counter value 0 and another counter value 2. This ensures uniqueness.

Another approach is to combine timestamps and counters in a 64-bit number. VoltDB generates unique ids dividing a 64-bit integer into sections, using 41 bits to identify a millisecond timestamp, 10 bits as a per-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. You can play with the bit sizes for each field as needed. Be very careful to anticipate and handle the case where time moves backward or stalls.

If you're looking for something conceptually simpler than getting incrementing ids correct, try an off-the-shelf 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 is described above. The upside is that it is safe to assume UUIDs are unique without much effort, but the downside is they often require 16 or more bytes to store.

Pattern: Use Kafka Offsets as Unique Identifiers

Unique identifiers are built-in when using Kafka. Combining the topic ID with the offset in the log can uniquely identify the event. This sounds like a slam dunk, but there are reasons to be careful.

1. Inserting items into Kafka has all of the same problems as any other distributed system. Managing exactly-once insertion into Kafka is not easy, and Kafka doesn't offer the right tools (at this time) to manage idempotency when writing to the Kafka topic.
2. If the Kafka cluster is restarted or switched, 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.

Example: Call Center Processing

Consider a customer support call center with two events:

1. A caller is put on hold (or starts a call).
2. A caller is connected to an agent.

The app must ingest these events and compute average hold time globally.

Version 1: Events Are Ordered

In this version, events for a given customer always arrive in the order in which they are generated. Event processing in this example is idempotent, so an event may arrive multiple times, but it can always be assumed that events arrive in the order they happened.

The schema for state 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 caller is put on hold (or a call is started), upsert a record into the set of ongoing holds. Use one of the methods described above to assign a unique id. Using an upsert instead of an insert makes this operation idempotent.

When a caller is connected to an agent, look up the corresponding ongoing hold in the state. If the ongoing hold is found, 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 space efficient. But is the key assumption valid? Can we guarantee order? The answer is that guaranteeing order is certainly possible, but it's hidden 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.

Version 2: Events Are Not Ordered

In this version, events may arrive in any order. The problem with unordered events is that you can't 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 any repeat messages.

How long do we need to keep these events? We must hold them until we're sure that another event for a particular hold could not arrive. This may be minutes, hours, or days, depending on your situation.

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

When to Avoid This Pattern

Idempotency can add storage overhead to store extra IDs for uniqueness. It can add a fair bit of complexity, depending on many factors, such as whether your event processor has certain features or whether your app requires complex operations to be idempotent.

Making the effort to build an idempotent application should be weighed against the cost of having imperfect data once in a while. It's important to keep in mind that some data has less value than other data, and spending developer time ensuring it's perfectly processed may be a poor allocation of resources.

Another reason to avoid idempotent operations is that the event processor or data store makes it very hard to achieve, based on the functionality you are trying to deliver, and switching tools is not a good option.

Related Concepts and Techniques

- Delivery Guarantees: discussed in detail in Chapter 5
- Exponential Backoff: defined in the Glossary.
- CRDTs: defined in the Glossary and referenced in "Idea in Brief" on page 27.
- ACID: defined and discussed in detail in Chapter 2.

Glossary

ACID
: See "What Is ACID?" on page 7.

Big Data
: The volume and variety of information collected. Big data is an evolving term that describes any large amount of structured, semi-structured, and unstructured data that has the potential to be mined for information. Although big data doesn't refer to any specific quantity, the term is often used when speaking about petabytes and exabytes of data. Big data systems facilitate the exploration and analysis of large data sets. VoltDB is not big data, but it does support analytical capabilities using Hadoop, a big data database.

CAP
: See "What Is CAP?" on page 9.

Commutative Operations
: A set of operations are said to be commutative if they can be applied in any order without affecting the ending state.

  For example, a list of account credits and debits is considered commutative because any ordering leads to the same account balance. If there is an operation in the set that checks for a negative balance and charges a fee, then the order in which the operations are applied absolutely matters.

CRDTs
: Conflict-free, replicated datatypes are collection data structures designed to run on systems with weak CAP consistency, often across multiple data centers. They leverage commutativity and monotonicity to achieve strong eventual guarantees on replicated state.

  Compared to strongly consistent structures, CRDTs offer weaker guarantees, additional complexity, and can require additional space. However, they remain available for writes during network partitions that would cause strongly consistent systems to stop processing.

Delivery Guarantees
: See Chapter 5.

Determinism
: In data management, a deterministic operation is one that

will always have the exact same result given a particular input and state. Determinism is important in replication. A deterministic operation can be applied to two replicas, assuming the results will match. Determinism is also useful in log replay. Performing the same set of deterministic operations a second time will give the same result.

Dimension Data

Dimension data is infrequently changing data that expands upon data in fact tables or event records.

For example, dimension data may include products for sale, current customers, and current salespeople. The record of a particular order might reference rows from these tables so as not to duplicate data. Dimension data not only saves space, it allows a product to be renamed and have that rename reflected in all open orders instantly. Dimensional schemas also allow easy filtering, grouping, and labeling of data.

In data warehousing, a single *fact table*, a table storing a record of facts or events, combined with many dimension tables full of dimension data, is referred to as a *star schema*.

ETL

Extract, transform, load is the traditional sequence by which data is loaded into a database. Fast data pipelines may either compress this sequence, or perform analysis on or in response to incoming data before it is loaded into the long-term data store.

Exponential Backoff

Exponential backoff is a way to manage contention during failure. Often, during failure, many clients try to reconnect at the same time, overloading a recovering system.

Exponential backoff is a strategy of exponentially increasing the timeouts between retries on failure. If an operation fails, wait one second to retry. If that retry fails, wait two seconds, then four seconds, etc,…. This allows simple one-off failures to recover quickly, but for more-complex failures, there will eventually be a low-enough load to successfully recover. Often the growing timeouts are capped at some large number to bound recovery times, such as 16 seconds or 32 seconds.

Fast Data

The processing of streaming data at real-time velocity, enabling instant analysis, awareness, and action. Fast data is data in motion, streaming into applications and computing environments from hundreds of thousands to millions of endpoints—mobile devices, sensor networks, financial transactions, stock tick feeds, logs, retail systems, telco call routing and authorization systems, and more.

Systems and applications designed to take advantage of fast data enable companies to make real-time, per-event decisions that have direct, real-time

impact on business interactions and observations.

Fast data operationalizes the knowledge and insights derived from "big data" and enables developers to design fast data applications that make real-time, per-event decisions. These decisions may have direct impact on business results through streaming analysis of interactions and observations, which enables in-transaction decisions to be made.

HTAP

Hybrid transaction/analytical processing (HTAP) architectures, which enable applications to analyze "live" data as it is created and updated by transaction-processing functions, are now realistic and possible.

From the Gartner 2014 Magic Quadrant: "…they must use the data from transactions, observations, and interactions in real time for decision processing as part of, not separately from, the transactions. This process is the definition of HTAP (for further details, see "Hype Cycle for In-Memory Computing, 2014"). Source: Gartner, Inc. Analyst: Massimo Pezzini; "Hybrid Transaction/Analytical Processing Will Foster Opportunities for Dramatic Business Innovation," January 2014.

Idempotence

An idempotent operation is an operation that has the same effect no matter how many times it is applied.

See Chapter 6 for a detailed discussion of idempotence, including a recipe explaining the use of idempotent processing.

Metadata

Metadata is data that describes other data. Metadata summarizes basic information about data, which can make finding and working with particular instances of data easier.

Operational Analytics (another term for operational BI)

Operational analytics is the process of developing optimal or realistic recommendations for real-time, operational decisions based on insights derived through the application of statistical models and analysis against existing and/or simulated future data, and applying these recommendations in real-time interactions.

Operational Database

Operational database management systems (also referred to as OLTP, or On Line Transaction Processing databases) are used to manage dynamic data in real time. These types of databases allow you to do more than simply view archived data. Operational databases allow you to modify that data (add, change, or delete) in real time.

Real-Time Analytics

Real-time analytics is an overloaded term. Depending on context, real-time means different things. For example, in many OLAP use cases, real-time can mean minutes or hours; in fast data use cases, it may mean milliseconds. In one sense, *real-time* implies that analytics can

be computed while a human waits. That is, answers can be computed while a human waits for a web dashboard or report to compute and redraw.

Real-time also may imply that analytics can be done in time to take some immediate action. For example, when a user uses too much of their mobile data plan allowance, a real-time analytics system can notice this and trigger a text message to be sent to that user.

Finally, real-time may imply that analytics can be computed in time for a machine to take action. This kind of real-time is popular in fraud detection or policy enforcement. The analysis is done between the time a credit or debit card is swiped and the transaction is approved.

Probabilistic Data Structures

Probabilistic data structures are data structures that have a probabilistic component. In other words, there is a statistically bounded probability for correctness (as in Bloom filters).

In many probabilistic data structures, the access time or storage can be an order of magnitude smaller than an equivalent non-probabilistic data structure. The price for this savings is the chance that a given value may be incorrect, or it may be impossible to determine the exact shape or size of a given data structure. However, in many cases, these inconsistencies are either allowable or can trigger a broader, slower search on a complete data structure. This hybrid approach allows many of the benefits of using probability, and also can ensure correctness of values.

Shared Nothing

A shared-nothing architecture is a distributed computing architecture in which each node is independent and self-sufficient and there is no single point of contention across the system. More specifically, none of the nodes share memory or disk storage.

Streaming Analytics

Streaming analytics platforms can filter, aggregate, enrich, and analyze high-throughput data from multiple disparate live data sources and in any data format to identify simple and complex patterns to visualize business in real time, detect urgent situations, and automate immediate actions (definition: Forrester Research).

Streaming operators include: Filter, Aggregate, Geo, Time windows, Temporal patterns, and Enrich.

Translytics

Transactions and analytics in the same database (source: Forrester Research).

About the Authors

Ryan Betts is one of the VoltDB founding developers and is presently VoltDB CTO. Ryan came to New England to attend WPI. He graduated with a B.S. in Mathematics and has been part of the Boston tech scene ever since. Ryan has been designing and building distributed systems and high-performance infrastructure software for almost 20 years. Chances are, if you've used the Internet, some of your ones and zeros passed through a slice of code he wrote or tested.

John Hugg, founding engineer & Manager of Developer Relations at VoltDB, specializes in the development of databases, information management software, and distributed systems. As the first engineer on the VoltDB product, he worked with the team of academics at MIT, Yale, and Brown to build H-Store, VoltDB's research prototype. John also helped build the world-class engineering team at VoltDB to continue development of the company's open source and commercial products. He holds a B.S. in Mathematics and Computer Science and an M.S. in Computer Science from Tufts University.