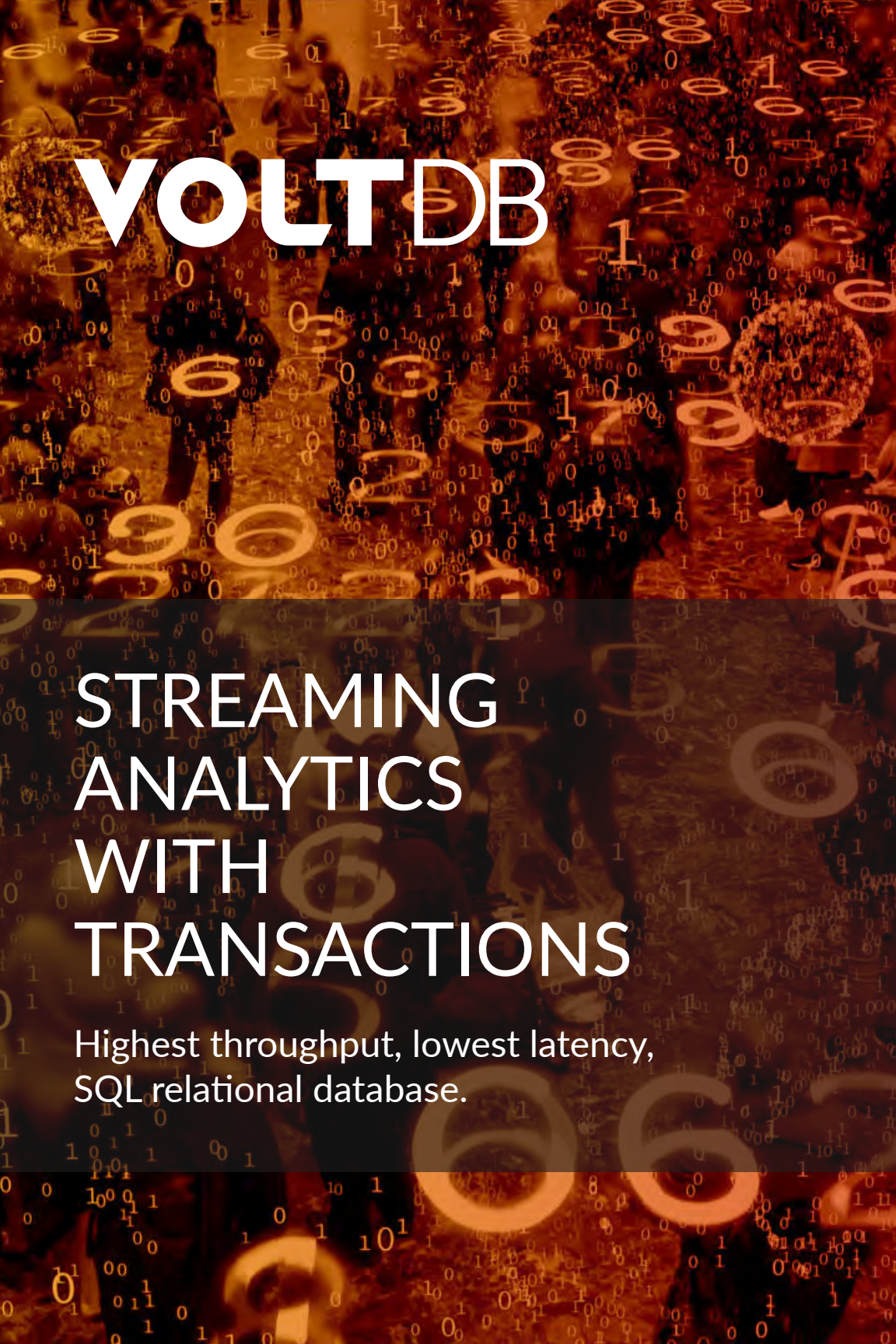
O'REILLY®

Compliments of VOLTDB

Fast Data Front Ends for Hadoop

Transaction and Analysis Pipelines

Akmal Chaudhri


VOLTDB
STREAMING ANALYTICS WITH TRANSACTIONS
Highest throughput, lowest latency, SQL relational database.

Fast Data Front Ends for Hadoop

Transaction and Analysis Pipelines

Akmal Chaudhri

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

Fast Data Front Ends for Hadoop
by Akmal Chaudhri

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

978-1-491-93779-2

[LSI]

Table of Contents

Fast Data Front Ends for Hadoop

Building streaming data applications that can manage the massive quantities of data generated from mobile devices, M2M, sensors, and other IoT devices is a big challenge many organizations face today.

Traditional tools, such as conventional database systems, do not have the capacity to ingest fast data, analyze it in real time, and make decisions. New technologies, such as Apache Spark and Apache Storm, are gaining interest as possible solutions to handling fast data streams. However, only solutions such as VoltDB provide streaming analytics with full Atomicity, Consistency, Isolation, and Durability (ACID) support.

Employing a solution such as VoltDB, which handles streaming data, provides state, ensures durability, and supports transactions and real-time decisions, is key to benefitting from fast (and big) data.

Data ingestion is a pressing problem for any large-scale system. Several architecture options are available for cleaning and preprocessing data for efficient and fast storage. In this report, we will discuss the advantages and disadvantages of various fast data front ends for Hadoop.

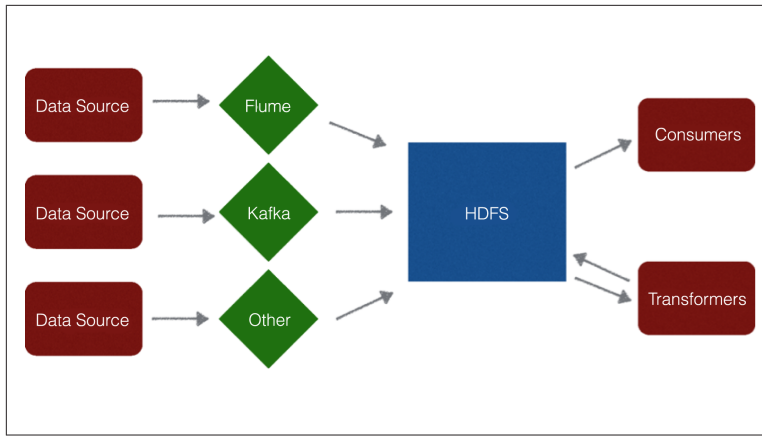


Figure 1-1. Typical big data architecture

Figure 1-1 presents a high-level view of a typical big data architecture. A key component is the HDFS file store. On the left-hand side of HDFS, various data sources and systems, such as Flume and Kafka, move data into HDFS. The right-hand side of HDFS shows systems that consume the data and perform processing, analysis, transformations, or cleanup of the data. This is a very traditional batch-oriented picture of big data.

All systems on the left-hand side are designed only to move data into HDFS. These systems do not perform any processing. If we add an extra processing step, as shown in Figure 1-2, the following significant benefits are possible:

1. We can obtain better data in HDFS, because the data can be filtered, aggregated, and enriched.
2. We can obtain lower latency to understanding what's going on with this data with the ability to query directly from the ingestion engine using dashboards, analytics, triggers, counters, and so on for real-time alerts. First, this allows us to understand things immediately as the data are coming in, not later in some batch process. In innumerable business use cases, response times in minutes versus hours, or even seconds versus minutes, make a huge difference (to say nothing of the growing number of life-critical applications in the IoT and the Industrial Internet). Second, the ability to combine analytics with transactions is a very powerful combination that goes beyond simple stream-

ing analytics and dashboards to provide intelligence and context in real time.

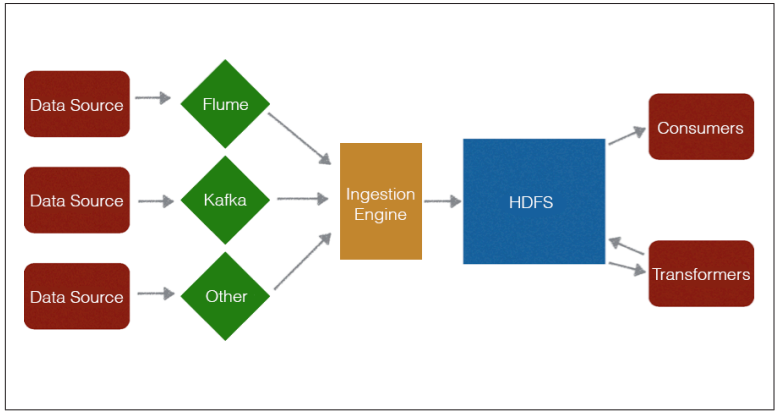


Figure 1-2. Adding an ingestion engine

Let's now discuss the ingestion engine, shown in Figure 1-2, in more detail. We'll begin with the three main value propositions of using an ingestion engine as a fast data front end for Hadoop.

Value Proposition #1: Cleaning Data

Filtering, de-duplication, aggregation, enrichment, and de-normalization at ingestion can save considerable time and money. It is easier to perform these actions in a fast data front end than it is to do so later in batch mode. It is almost zero cost in time to perform these actions at ingestion, as opposed to running a separate batch job to clean the data. Running a separate batch job requires storing the data twice—not to mention the processing latency.

De-duplication at ingestion time is an obvious example. A good use case would be sensor networks. For example, RFID tags may trip a sensor hundreds of times, but we may only really be interested in knowing that an RFID tag went by a sensor once. Another common use case is when a sensor value changes. For example, if we have a temperature sensor showing 72 degrees for 6 hours and then suddenly it shows 73 degrees, we really need only that one data point that says the temperature went up a degree at a particular time. A fast data front end can be used to do this type of filtering.

A common alternative approach is to dump everything into HDFS and sort the data later. However, sorting data at ingestion time can provide considerable benefits. For example, we can filter out bad data, data that may be too old, or data with missing values that requires further processing. We can also remove test data from a system. These operations are relatively inexpensive to perform with an ingestion engine. We can also perform other operations on our data, such as aggregation and counting. For example, suppose we have a raw stream of data arriving at 100,000 events per second, and we would really like to send one aggregated row per second to Hadoop. We filter by several orders of magnitude to have less data. The aggregated row can pick from operations such as count, min, max, average, sum, median, and so on.

What we are doing here is taking a very large stream of data and making it into a very manageable stream of data in our HDFS data set. Another thing we can do with an ingestion engine is delay sending aggregates to HDFS to allow for late-arriving events. This is a common problem with other streaming systems; events arrive a few seconds too late and data has already been sent to HDFS. By preprocessing on ingest, we can delay sending data until we are ready. Avoiding re-sending data speeds operations and can make HDFS run orders of magnitude faster.

Consider the following real-life example taken from a call center using VoltDB as its ingestion engine. An event is recorded: a call center agent is connected to a caller. The key question is: "How long was the caller kept on hold?" Somewhere in the stream before this event was the hold start time, which must be paired up with the event signifying the hold end time. The user has a Service Level Agreement (SLA) for hold times, and this length is important. VoltDB can easily run a query to find correlating events, pair those up, and push those in a single tuple to HDFS. Thus, we can send the record of the hold time, including the start and duration, and then later any reporting we do in HDFS will be much simpler and more straightforward.

Another example is from the financial domain. Suppose we have a financial application that receives a message from the stock exchange that order 21756 was executed. But what is order 21756? The ingestion engine would have a table of all outstanding orders at the stock exchange, so instead of just sending these on to HDFS, we could send HDFS a record that 21756 is an order for 100 Microsoft

shares, by a particular trader, using a particular algorithm and including the timestamp of when the order was placed, the timestamp it was executed, and the price the shares were bought for.

Data is typically de-normalized in HDFS even though it may be normalized in the ingestion engine. This makes analytic queries in HDFS much easier; its schema-on-read capability enables us to store data without knowing in advance how we'll use it. Performing some organization (analytics) at ingestion time with a smart ingestion engine will be very inexpensive in both time and processing power, and can have a big payoff later, with much faster analytical queries.

Value Proposition #2: Understanding

Value proposition #2 is closely related to the first value proposition. Things we discussed in value proposition #1 regarding storing better quality data into HDFS can also be used to obtain a better understanding of the data. Thus, if we are performing aggregations, we can also populate dashboards with aggregated data. We can run queries that support filtering or enrichment. We can also filter data that meets very complex criteria by using powerful SQL queries to understand whether data is interesting or not. We can write queries that make decisions on ingest. Many VoltDB customers use the technology for routing decisions, including whether to respond to certain events. For example in an application that monitors API calls on an online service, has a limit been reached? Or is the limit being approached? Should an alert be triggered? Should a transaction be allowed? A fast data front end can make many of these decisions easily and automatically.

Business logic can be created using a mix of SQL queries and Java processing to determine whether a certain condition has been met, and take some type of transactional action based upon it. It is also possible to run deep analytical queries at ingestion time, but this is not necessarily the best use for a fast data front end. A better example would be to use a dashboard with aggregates. For example, we might want to see outstanding positions by feature or by algorithm on a web page that refreshes every second. Another example might be queries that support filtering or enrichment at ingestion—seeing all events related to another event and determining if that event is the last in a related chain in order to push a de-normalized enriched tuple to HDFS.

Value Proposition #3: Decision Making

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 what 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 so much 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.

One Solution In Depth

Given the goals we have discussed so far, we want our system to be as robust and fault tolerant as possible, in addition to keeping our data safe. But it is also really important that we get the correct answers from our system. We want the system to do as much work for the user as possible, and we don't want to ask the developers to write code to do everything. The next section of this report will examine VoltDB's approach to the problems of pre-processing data and fast analytics on ingest. VoltDB is designed to handle the hard parts of the distributed data processing infrastructure, and allow developers to focus on the business logic and customer applications they're building.

So how does this actually work when we want to both understand queries and process data? Essentially because of VoltDB's strong ACID model, we just write the logic in Java code, mixed with SQL. This is not trivial to do, but it is easier because the state of the data and the processing are integrated. We also don't have to worry about system failure, because if the database needs to be rolled back, we have full atomicity.

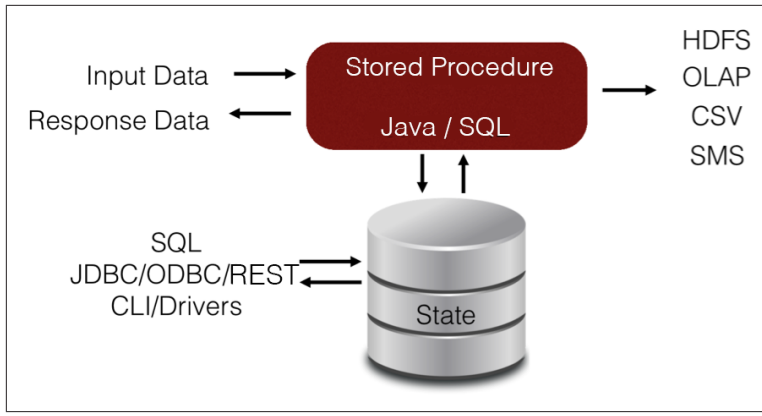


Figure 1-3. VoltDB solution

In Figure 1-3, we have a graphic that shows the VoltDB solution to the ingestion engine discussed earlier. We have a stored procedure that runs a mix of Java and SQL, and it can take input data. Something that separates VoltDB from other fast data front end solutions is that VoltDB can directly respond to the caller. We can push data out into HDFS; we can also push data out into SQL analytics stores, CSV files, and even SMS alerts. State is tightly integrated, and we can return SQL queries, using JDBC, ODBC, even JavaScript over a REST API. VoltDB has a command-line interface, and native drivers that understand VoltDB's clustering. In VoltDB, state and processing are fully integrated and state access is global. Other stream-processing approaches, such as Apache Storm, do not have integrated state. Furthermore, state access may or may not be global, and it is disconnected. In systems such as Spark Streaming, state access is not global, and is very limited. There may be good reasons to limit state access, but it is a restricted way to program against an input stream.

VoltDB supports standard SQL with extensions. It is fully consistent, with ACID support, as mentioned earlier. It supports synchronous, inter-cluster High Availability (HA). It also makes writing applications easier because VoltDB supports native aggregations with full, SQL-integrated, live materialized views. Users can write a SQL statement saying "maintain this view as my data changes." We can query that view in milliseconds. Also available are easy counting, ranking, and sorting operations. The ranking support is not just the top 10, for example. We can also perform ranking such as "show me the 10

people who are above me and behind me." VoltDB also uses existing Java libraries.

Bonus Value Proposition: The Serving Layer

We can connect Hadoop directly to VoltDB using SQL. This is essential, since we cannot easily get real-time responses with high concurrency from HDFS. Systems designed to query HDFS are not designed to run thousands or hundreds of thousands of requests per second. We cannot directly query Kafka or Flume, as these tools are not designed to move data. So querying our fast data front end makes perfect sense. VoltDB enables us to build a fast data front end that uses the familiar SQL language and standards. SQL is widely used today, and many companies have standardized on it. Some NoSQL database vendors also have embraced SQL to varying degrees.

Resilient and Reliable Data Front Ends

Having discussed the value that a fast data front end can provide, it's important to look at the theoretical and practical problems that can come up in an implementation. First, how wrong can data be? Delivery guarantees are a primary check on data reliability. Delivery guarantees typically fall into one of the following four categories:

1. At least once
2. At most once
3. None of the above
4. Exactly once

At-least-once delivery occurs when the status of an event is unclear. If we cannot confirm that a downstream system received an event, we send it again. This is not always easy to do because the event source needs to be robust. We cannot remove events until we have confirmed they have been processed by a downstream system, so it's necessary to buffer events for a little while and then resend them if it cannot be confirmed they have been received. Kafka makes this a lot easier; even so, it's easy to get wrong, and it's important to read the Kafka documentation to ensure this is being done correctly.

At-most-once delivery sends all events once. If something fails, some per-event data may be lost. How many events are lost is not always clear.

None-of-the-above is common, but users must be aware of the risks of duplicate data, missed data, or wrong answers. In this scenario, using Kafka as a high-level consumer, developers can keep track of roughly where they are in the stream, but not necessarily exactly. This periodically commits the offset pulled from Kafka, but doesn't necessarily track which offset was pushed into the downstream system. Using a Kafka high-level consumer, it is possible that we could have read more from Kafka or less from Kafka than we put into the downstream system, depending on how things are working. We could use one of Kafka's APIs if we really want to get something closer to exactly-once delivery. Any time two systems are involved in processing data, it may be difficult to get exactly-once delivery. The lesson here is that, even if we think we have at-least-once delivery, we might also have none-of-the-above.

Our final option is *exactly-once* delivery. There are two ways to get exactly-once delivery or very, very close to exactly-once for all intents and purposes:

1. If we have a system that has no side effects, we can use strong consistency to track incremental offsets, and if the processing is all atomic, then we should be able to get exactly-once delivery.
2. We can use at-least-once delivery but ensure everything is idempotent. This option is not exactly once. However, the effect is the same. An idempotent operation is one that has the same effect whether it is applied once or many times. If you are unsure whether your idempotent operation succeeded or failed after a failure, simply resend the operation; whether it ran once or twice, your data will be in the desired state.

These two methods for getting effectively exactly-once deliveries are not mutually exclusive. We can mix and match.

What are the best practices that come from this? Do we have a rough idea of how many events we should have processed? Can we check our dashboard to see how many we actually processed, and see if there is a discrepancy that we can understand? We have to assume, in this case, that our dashboard is correctly reflecting the numbers of events processed, which is not always easy to do. One solution to this problem is the Lambda Architecture, where everything is done twice—once in the fast, streaming layer in the ingestion engine, and once in the batch layer. We can use a Lambda Architecture to ensure exactly-once delivery, by checking in the

batch-processed pipeline for duplicates and gaps—but that means a latency-gap between batches before we're sure that exactly-once has been achieved. We can know tomorrow how wrong we were today, in short. That is certainly better than not understanding how wrong we were, but it is far from perfect.

Let's consider partial failure, where a process or update only half finishes. Managing partial failure is the first element of ACID transactions in a nutshell: atomicity. Atomicity means that the operation we asked for either completely finished or it didn't.

Let's look to Shakespeare for a simple example of atomicity. In *Romeo and Juliet,* Juliet plans to fake her death, and she sends someone to tell Romeo. Unfortunately, no one tells Romeo in time, and he believes that Juliet is actually dead, taking his own life in consequence. If this had been an ACID transaction, these two things would have happened together: Juliet would never have faked her death without notifying Romeo. Another example is from the movie *Dr. Strangelove*. Creating a doomsday device only works if, at the moment it's activated, you've told your enemy. Doing one of these things and not the other can cause problems.

In our earlier call center example when "call hold" ends, we want to remove the outstanding hold record. We want to add a new completed hold record. We want to mark an agent as busy. We want to update the total hold time aggregates and, actually, we want to do all four of these things. If an operation fails, it is much easier to just roll back and then try to do it again with some fixes. In our earlier financial example, when an order executes we want to remove it from the outstanding orders table. We want to update our position on Microsoft shares. We also want to update the aggregates for the trader, for the algorithm, and for the time window so the dashboards people are looking at are correct. Then we don't lose money.

Side Effects

Let's discuss the side effects of some popular ingestion engines in more detail. Essentially, any time we are performing an action in one system, it may trigger an action in another system. It is important to note that in many cases, stream-processing systems do not have integrated state, which can cause problems. Consider an example where we have a Storm processing node and we are writing to Cassandra for every event that comes to the Storm node. The problem

is that Storm and Cassandra are not well integrated. Storm does not have control over Cassandra. If the Storm node fails and we have written to Cassandra, or if the Storm node fails and we haven't written, do we know if we have or have not written to both? How do we recover from that? Certainly, Storm isn't going to roll back Cassandra if it has done the writing. We have to either accept that we may have some wrong answers, or make our application idempotent, meaning that we can do the same thing over repeatedly and obtain the same results, or we just accept that our answer is going to be wrong. Also, if Storm failed and the write is retried, perhaps the Cassandra write happens twice. If the Cassandra write fails, does the Storm operation fail? What happens if Storm retries and Cassandra fails repeatedly? The lack of tight integration may cause many problems.

Spark Streaming may also cause unwanted side effects. Spark Streaming, properly speaking, processes micro-batches, and it deals with native integration where it writes these micro-batches into a complete HDFS result that persists. If a Spark Streaming micro-batch fails, and it hasn't written a complete HDFS result set for that micro-batch, it can be rolled back. This tight integration is how Spark Streaming solves the problems of integrating Storm with Cassandra. By setting restrictions on how the state can be managed, the user gets a system that's much better at exactly-once processing, with better failure-handling properties.

The downside of this is that, because we're doing things in batches, and because when things fail we retry the batches, the latency for Spark Streaming is dramatically higher than the latency for Storm. It varies depending on what we're doing, but it could be orders of magnitude difference. Another issue is that we have state restrictions. We cannot write from Spark Streaming into Cassandra or Redis. We have to write from Spark Streaming into an HDFS result set. This isn't to say we couldn't write from Spark Streaming and other systems, but we would lose all the exactly-once properties as soon as we involve two systems that aren't tightly integrated.

What are the consequences? If we need to use two different systems and we want anything close to correctness under failure, we have two options:

1. Use two-phase commit between the two different systems.

   This is similar to what VoltDB does when it pushes data into the downstream HDFS system, OLAP store, or files on disk. What we can do is to buffer things, wait for confirmation from the downstream system, and then only delete things when we get the confirmation from the downstream system.

2. Use Kafka with Replay Smart.

   If we write things into Kafka, and then the consumer system fails, it can calculate where it actually failed and pick up from that point.

However, both of these options are difficult to get right, as they require more thought, planning, and engineering time.

What are the consequences from these things? When we're dealing with fast data, integration is key. This is really counterintuitive to people coming from the Hadoop ecosystem. If we are batch processing, integration is less important, because HDFS provides a safety net of immutable data sets where if a batch job fails and it only creates a partial result set, we can just remove it and start over again. By accepting batch, we have already accepted high latency, so it doesn't matter, or at least it's less inconvenient. Because the data sets in HDFS are largely immutable, we don't have as much risk that something is going to fail and lose data. With fast data, our data are actually moving around. The data's home is actually moving between the processing engine, the state engine, and into HDFS. The fewer systems we have, the more tightly integrated these systems are, and the better off we are going to be on the fast data side, which is very different than the batch side.

In summary, there are many benefits to adding processing at the ingestion phase of a Hadoop ecosystem. VoltDB provides a tightly integrated ingest engine, in addition to streaming analytics and in-transaction decisions. VoltDB also offers strong and easy-to-implement ACID compliance.

About the Author

Akmal B. Chaudhri is an Independent Consultant, specializing in big data, NoSQL, and NewSQL database technologies. He has previously held roles as a developer, consultant, product strategist, and technical trainer with several Blue-Chip companies and big data startups. He has regularly presented at many international conferences and served on the program committees for a number of major conferences and workshops. He has published and presented on emerging technologies and edited or co-edited ten books. He is now learning about Apache Spark and also how to become a Data Scientist. He holds a BSc (1st Class Hons.) in Computing and Information Systems, MSc in Business Systems Analysis and Design, and a PhD in Computer Science. He is a Member of the British Computer Society (MBCS) and a Chartered IT Professional (CITP).