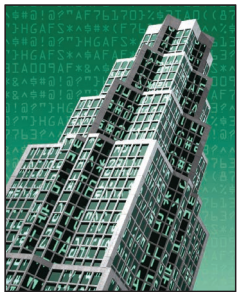
Editor: Jimmy Lin • jimmylin@uwaterloo.ca



The Lambda and the Kappa

Jimmy Lin • *University of Waterloo*

Big Data Bites

"Big Data Bites" is a regular department in *IEEE Internet Computing* that aims to deliver thought-provoking and potentially controversial ideas about all aspects of big data. Interested in contributing? Drop me a line!

—*Jimmy Lin*

Software architects designing solutions to real-world problems often face a choice between deploying the right tool for the job or a one-size-fits-all hammer. This decision is heavily influenced by the prevailing fashions of the day, and the broader arc of technological progress can be characterized as a pendulum swinging between the extremes. Like a pendulum, the motion continues endlessly without resolution, but unlike an actual pendulum, this movement can be erratic and unpredictable.

In this article, I reflect on modern architectures for integrating real-time and batch data analytics, exemplified by the lambda and kappa architectures. My high-level message is simple: there's no free lunch. It's not possible to have your cake and eat it too. If vendors, bloggers, evangelists, or anyone else claims otherwise, they're either lying, uninformed, or both. *Caveat emptor.*

Why? All software architectures capture tradeoffs, and deciding between alternatives is about choosing which tradeoffs you can live with in a particular context. However, the acceptability of tradeoffs changes over time — and that's why the pendulum continues to swing back and forth.

Warmup

As a warmup, let me briefly recap a parallel development in the relational database management system (RDBMS) world. When database luminary Michael Stonebraker famously declared that the one-size-fits-all database was an idea whose time had come and gone,[1] he was summarizing a trend that had already played out over the preceding decade or so. The pendulum had swung toward deploying the right tool for the job: In 1993, Edgar F. Codd and associates coined the term *online analytical processing* (OLAP) to describe a class of queries that were fundamentally different from online transaction processing (OLTP).[2] By the late 1990s, today's reference data warehouse architecture — distinct OLTP and OLAP databases connected by extract-transform-load (ETL) pipelines — was already well known.[3]

However, developers soon began to realize that ETL pipelines were difficult to build and maintain. I spent two years at Twitter (from 2010 to 2012) wrestling with this part of the data stack; for some personal experiences, see other work.[4–6] Another issue was that ETL pipelines introduced latency — nightly jobs (the norm) meant that business intelligence was being conducted on day-old data. As the pace of business quickened, organizations began demanding fresher and fresher data to guide decision making.

Cranking up the frequency of ETL (to hourly, say) was an obvious solution, but it merely stressed rickety ETL pipelines even more, often past the breaking point. For one account, see the work by Gilad Mishne and colleagues.[7] In the past few years, people began to explore alternative architectures — if we could perform analytical as well as transactional processing in the

 Published by the IEEE Computer Society 1089-7801/17/$33.00 © 2017 IEEE

Rant 1: I Hate the Name Lambda Architecture

I hate the name *lambda architecture* for two reasons. First, Nathan Marz described the lambda architecture in a blog post titled "How to Beat the CAP Theorem" (although that post doesn't actually coin the term, which came later).[1] The connection to the CAP theorem is, quite simply, nonsensical.

Second, the post reeks of (typical Silicon Valley) hubris. He writes:

> This post is going to challenge your basic assumptions on how data systems should be built. But by breaking down our current ways of thinking and reimagining how data systems should be built, what emerges is an architecture more elegant, scalable, and robust than you ever thought possible.[1]

As if this way of designing systems had never been thought of before! Just as a point of reference, Turing Award winner Butler Lampson describes in a 1983 paper[2] exactly what we would today call the lambda architecture, providing an example from — of all organizations known for technological innovations — Bank of America! Lampson's paper outlines a bunch of design principles, which meant that if the lambda architecture was already well known over three decades ago, its invention surely must have been much earlier.

How silly is this development? It's as if I had rediscovered the wheel, called it the *omicron device*, wrote a blog post about the omicron device being the solution to beating friction, and then for some reason (gullibility of the clickbait-driven tech press? the Twitter-verse echo chamber?) others began picking it up. Finally, once critical mass has been achieved, everyone starts saying (with a straight face), "our vehicles employ the omicron device for superior conveyance capabilities." And no one is around to call out the emperor's lack of clothes.

The sheer ridiculousness of this state of affairs baffles me, but perhaps this makes me now an old, curmudgeonly academic? I hate the name lambda architecture, but unfortunately, we appear to be stuck with it.

References

1. N. Marz, "How to Beat the CAP Theorem," *Thoughts from the Red Planet*, blog, 13 Oct. 2011; http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html.
2. B.W. Lampson, "Hints for Computer System Design," *Proc. 9th ACM Symp. Operating Systems Principles*, 1983; doi:10.1145/800217.806614.

same database, everything would be hunky dory, right? (Short answer: it's an improvement but not a panacea; don't buy the hype completely.) Regardless, we've seen the emergence of hybrid transactional/analytical processing (HTAP), a brilliant term invented by Gartner in 2014.

The pendulum has now swung the other way, back toward one-size-fits-all solutions.

Lambda

Our short detour into HTAP is relevant to this column because the underlying pain point is the same. What's the problem that organizations are trying to solve today? Real-time analytics for insight generation. Social media companies want to adapt ad placement algorithms in real time, financial institutions want to stop fraud as it's being committed, and municipalities want to respond to incidents with minimal delay, just to give a few examples. Of course, organizations want real-time capabilities without sacrificing analytics over large volumes of historic data.

Because this column derives from personal experience, unavoidably it will be colored by my having worked at Twitter on a number of projects that tackle exactly this challenge. Of course it will be a biased perspective, but on the flip side, I can speak directly from first-hand knowledge.

What's the solution? One established approach is the so-called lambda architecture, proposed by Nathan Marz of a company called BackType (circa 2011). Distilled to its essence, the lambda architecture consists of a batch processing layer (platform) and a transient real-time processing layer (platform), plus a merging layer on top.[8]

Consider a concrete example: say we wanted to count tweet impressions. Furthermore, not only do we want real-time updates as users are tapping, swiping, and clicking *right now*, but we want historic counts dating back to the moment a tweet was posted (for example, consider a tweet by Donald Trump last year that's receiving a new burst of engagement).

In Twitter, impression data are captured by frontend logs. After a multistage aggregation pipeline, the logs are deposited into a Hadoop data warehouse (a data lake, in modern parlance) — see the work by George Lee and colleagues for details.[5] In the lambda architecture at Twitter, circa 2012, the batch processing layer was MapReduce, where such aggregations were already routine. However, the logging pipeline introduced a delay: even in the best possible case, logs were a few hours old (see the previous reference[5] for a discussion of why these latencies are unavoidable). This meant that a dashboard of tweet

impressions powered by MapReduce alone would always be a few hours out of date. Not good enough!

Ah, but that's when the real-time layer comes in — which in this case was Storm, a framework for real-time processing that provides a dataflow graph abstraction (vertices represent computations and edges capture data movement). Storm performed real-time aggregations, and a merging layer encapsulated the logic to integrate batch and real-time results. Once the (delayed) results from the batch layer arrived, results from the real-time layer could be discarded. In other words, the batch computations provided truth, while the real-time results were transient.

Great solution, huh? Well, Twitter thought so — so much so that it acquired BackType, the company that developed Storm, in 2011.

With the lambda architecture, all our problems are solved, right? Not quite so fast — yes, the lambda architecture bestows new capabilities, but there's no free lunch. These new capabilities come at a significant cost.

In this case, the cost is complexity. The lambda architecture basically means that everything must be written twice: once for the batch platform and again for the real-time platform. In many cases, the implementations are completely different.

For example, in batch processing it's easy to compute the cardinality of a set (for example, the number of tweets in a particular hour), and this value can be used for a variety of computations and optimizations. This isn't possible in a real-time platform if you're processing input incrementally. Another example: in batch processing, you often don't need to worry about a particular dictionary growing larger than the amount of memory available because the framework will automatically spill to disk. Not so on a real-time platform: if your dictionary overflows memory, bad things will happen.

It gets worse: two separate implementations need to be indefinitely maintained in parallel, sometimes by separate teams. This means that changes need to be propagated from one to the other, or else the final results will be suspect. The real-time platform just updated the machine-learned model for detecting fraudulent clicks? Better make sure the same model is applied in the batch processing layer. Oh, wait, the model has a blacklist that is updated continuously? Uh, how are we going to backfill historical data?

We're not done yet: even when the lambda architecture is working as intended, the semantics of the computations are unclear. As a concrete example, aggregate values can sometimes fluctuate unpredictably. Let's say the Storm cluster suffered a transient load spike and 10 minutes' worth of log data got dropped. No one would notice this until the logs are processed by the batch layer some time later. Logging pipelines typically form a different code path than the real-time processing layer, and are usually more robust because persistence is an explicit design goal. In this scenario, the missing data reappear and the aggregate values change suddenly. Ah, but of course, we can fix this by building a better streaming platform — one with more precise semantics. Yes, hold that thought, we'll come back to it later.

With the lambda architecture, we've gained new capabilities, but at a cost in increased complexity. There's no free lunch. However, and this is the key point, the tradeoff was worthwhile for the needs of Twitter at the time. The adoption of the lambda architecture can be heralded as a success, even given its deficiencies.

The next evolution of the lambda architecture at Twitter began around late 2012. A bunch of smart engineers wondered if it would be possible to develop an abstraction to integrate online and batch MapReduce computations within a single framework. The complete story is captured in a 2014 paper,[4] but to make a long story short, Summingbird is a domain-specific language (DSL) that allows queries to be automatically translated into MapReduce jobs or Storm topologies.

Neat, right? But there's a catch: the key insight of Summingbird is that certain algebraic structures provide the theoretical foundation for seamlessly integrating batch and online processing. In practice, this means that all aggregations must be expressed (somewhat simplifying) as commutative monoids. In other words, the tradeoff is expressiveness (no arbitrary computations) for simplicity (not needing separate batch and real-time implementations).

Nevertheless, this tradeoff was worthwhile, because Summingbird captured common cases that the lambda architecture was designed for. TimeSeries AggregatoR (TSAR) was the next step in this evolution, trading off even more expressiveness for even greater simplicity.[9]

The lambda architecture exemplifies deploying the right tool for the job — a platform for batch processing and a separate platform for real-time processing — and then dealing with the complexities of munging two sets of results together. But, as Summingbird illustrates, the pendulum began to swing back again ...

Kappa

Enter the kappa architecture, proposed in a 2014 blog post by Jay Kreps,[10] one of the original authors of Kafka and a data architect at LinkedIn at the time. In the kappa architecture, everything's a stream. And if everything's a stream, all you need is a stream processing engine. What the lambda architecture would call batch processing is simply streaming through historic data. Kreps writes, somewhat tongue in cheek, "Maybe we could call this the Kappa Architecture, though it

Rant 2: Everyone Is Sloppy about Processing Semantics

The interwebs abound with discussions of at-least-once, at-most-once, and exactly-once semantics in the context of stream processing. Unfortunately, nearly everyone is sloppy about the use of these terms, including myself. In the latest iteration of my "Big Data Infrastructure" course (see https://lintool.github.io/bigdata-2017w), I was called out by my colleague Keshav for being imprecise on exactly this point in lecture.

The fundamental issue is that messaging semantics were formulated primarily to describe communication channels — that is, getting data from point A to point B. For example, from the end-to-end perspective, TCP provides exactly-once semantics (with a few caveats). I send these bytes through the pipe, and the exact same bytes appear at the other end. This is accomplished with sequence numbers, acknowledgments, and deduplication of packets. The last point is critical — it's not uncommon for duplicate packets to arrive at a destination (what we might term at-least-once packet delivery), but TCP transparently handles deduplication to correctly reconstruct the original message.

Trouble arises when we try to extend semantics about communications channels to semantics about computations. In the case of the dataflow graph model, what does exactly-once refer to? Does it refer to the delivery of messages along edges in the computation graph? What about the case where a message is successfully delivered to the incoming queue of a computation vertex — some other process dequeues the message and then the machine crashes before the message has been processed? What happens? Or consider the symmetric case when a processed message has been deposited successfully in the outgoing queue, but the machine crashes before the message has been sent to the next processing vertex?

One possible (more precise) definition of exactly-once processing is that the computation represented by each vertex in the dataflow graph is applied exactly once to each message that enters the computation graph (in the correct order, as determined by the edges of the graph). This escape hatch appeals to the notion of the correct answer, as opposed to over-counting or under-counting in the case of simple aggregations.

However, we're not out of the woods yet — even the correct answer can be difficult to define. Consider the simple case where we have a continuous computation that monitors load on a number of servers to identify the one with the heaviest load. Let's say server A, with the heaviest load, crashed before the computation completed. Because A crashed, its load is obviously zero. But our system answers "A." Is the answer correct?

Well, we could say, the semantics of the correct answer is when the computation began — but then we have to appeal to clock synchronization and a host of other thorny issues … TrueTime[1] to the rescue?

Regardless, my point is that the terms at-least-once, at-most-once, and exactly-once are tossed around with such sloppiness as to render meaningful technical discussions difficult. I don't have any concrete suggestions, except to encourage that we all be more precise in our characterizations.

Reference

1. J.C. Corbett et al., "Spanner: Google's Globally-Distributed Database," *Proc. 10th Usenix Symp. Operating System Design and Implementation*, 2012, pp. 251–264.

may be too simple of an idea to merit a Greek letter."[10]

The initial execution of this vision was a processing framework called Samza, which used Kafka as the underlying messaging fabric. The most recent incarnation is Kafka Streams,[11] which is a lightweight library on top of Kafka. The story goes like this: we can think of a stream as an append-only collection, and well, that's a log. A table is merely the cache of the latest value of each key in the log, and the log is a record of each update to the table.

Kafka is essentially a distributed log, so the logging concept has been there since day one. Kafka Streams adds the table abstraction as a first-class citizen, cleverly implemented as compacted topics. Thus, Kafka elegantly captures this log/table duality. To a database person, this all sounds suspiciously familiar, like incremental view maintenance? Yup – and to his credit, Kreps fully acknowledges this as a "slight rebranding."

So, the kappa architecture represents a swing of the pendulum back to a one-size-fits-all solution. And in fact, Kafka wasn't even the earliest example. Spark[12] can perhaps be characterized (once again, tongue-in-cheek) as the anti-kappa architecture, in that everything is batch. Its basic abstraction is the discretized stream: let's chop up the stream into itsy-bitsy batches and reuse the Spark machinery for batch processing. Don't worry about streams, they're just mini-batches!

Another related development is the dataflow model of Tyler Akidau and colleagues,[13] which was packaged as Google Cloud Dataflow and later donated to the open source community to serve as the basis of Apache Beam. This model traces its origins back to Google's MillWheel system[14] for streaming, low-latency computations.

Apache Beam presents a rich API that explicitly recognizes the difference between event time, the time when an event actually occurred, and processing time, the time when the event is observed in the system (a distinction, for example, not captured in Spark Streaming, but recently introduced in Spark's Structured Streaming API). For example, an event occurring at 4:17 (event time) isn't observed until 4:20 (processing time) due to delays in the logging pipeline.

The notion of a watermark captures the relationship between the two and tries to make a statement about the completeness of observed data with respect to event times. For example, a heuristic watermark might capture the fact that 99 percent of messages will arrive within 5 minutes of the event time (based, for example, on analyses of historic data). Watermarks determine triggers for moving window aggregations — in our example, it would make sense to compute the aggregation 5 minutes after the event time, when we'd expect 99 percent of the messages to have arrived.

However, this isn't enough — in some cases, we can't wait 5 minutes, and it'd be nice to have some early results (that is, early triggerings). Furthermore, because watermarks are heuristic in many cases, events might still arrive after the trigger (1 percent of the events, in fact, in this example), so these late events need to be handled somehow. The Apache Beam API provides one possible abstraction for managing these complexities.

Interestingly, the Beam model doesn't distinguish between batch and streaming computations. Rather, the primary dichotomy is between bounded and unbounded datasets. What we would traditionally call batch processing is simply streaming through bounded datasets. In short, this represents an instance of the kappa architecture!

In contrast to the lambda architecture, the kappa architecture promotes the elegance of a one-size-fits-all solution. But there's always a catch! In this case, there's the simple fact that a streaming engine will never be as fast or as efficient as a batch processing engine with the same capabilities and implementation quality. There will always be a latency/throughput tradeoff.

As a start, processing semantics are much more straightforward with batch processing: Hadoop distributed file system blocks for MapReduce and resilient distributed datasets (RDDs) in Spark are both immutable. As Pat Helland declares, "immutability changes everything!"[15]

For a streaming engine that adopts a dataflow graph abstraction, to ensure that each vertex in the graph receives each message at least once, we must have some type of acknowledgment mechanism. There is unavoidable overhead associated with this, and if we want minimal latency, we have to pay the tremendous cost of message-by-message acknowledgment. Of course, we could back off to acknowledgment of *batches* of messages (a standard optimization), but then we're sacrificing latency.

Sure, there exists a continuum between message-by-message incremental processing and batch processing, and indeed Spark Streaming's discretized streams represent a mini-batching middle ground, but my overall point remains: there's no free lunch. In the case of Spark Streaming, there's a floor on minimum latency (on the order of a few hundred milliseconds), but that's okay — this meets the latency requirements of most applications anyway.

Summarizing, the kappa architecture isn't necessarily better or worse than the lambda architecture. It simply makes a different set of tradeoffs. In fact, Kreps writes in his original blog post: "The real advantage isn't about efficiency at all, but rather about allowing people to develop, test, debug, and operate their systems on top of a single processing framework."[10] A stream processing engine (the one-size-fits-all hammer) is never going to be faster than a batch processing engine (the right tool for the job) on a batch processing task. The question is, how much is the performance penalty, and are we willing to give it up to regain the sanity of not having to write everything twice?

Lunch

The message of this column is simple: there's no free lunch in the quest for analytics on both historic data and in real time. The lambda and the kappa architectures characterize the right tool for the job and the one-size-fits-all approaches, respectively. They capture different tradeoffs in terms of desiderata such as low latency, high throughput, low cost, precise semantics, easy maintenance, and so on. It's not possible to have it all, and the ultimate choice depends on a particular organization's sensitivity to each of these features.

To conclude, I'd like to discuss why the pendulum continues to swing back and forth endlessly. At a high level, deploying the right tool for the job can be characterized as increased specialization, which has a tendency to increase integration costs when putting components together to solve a particular problem. At one apex of the pendulum swing, gains from increased specialization become lost in the pain of integration. The pendulum then starts swinging back, and architects attempt to simplify the toolchain by increased generalization, relinquishing the gains that come from specialization.

At the other apex of the pendulum swing, we arrive at a one-size-fits-all solution. Architects eventually realize what this actually means — not perfect for anything, but good enough for most things.[16] Typically, this realization is accompanied by the need

for new capabilities that stretch the single hammer to the breaking point, which forces architects to consider specialized tools, thus beginning the cycle anew.

Perhaps there's a way out of this quagmire? Computer scientists have one incredibly powerful tool to manage complexity: abstraction. This is illustrated by Summingbird, which I introduced previously as an example of the pendulum swinging back toward one size fits all. However, a more precise characterization is that the Summingbird DSL aims toward generalization at the API level, while leaving the implementations specialized. Can we have our cake and eat it too?

Computer scientists have one incredibly powerful tool to manage complexity: abstraction.

This is where the lambda/kappa labels start to break down and it's more useful to discuss generalized versus specialized approaches. Just like in Summingbird, abstraction serves a powerful role in Beam, by providing a unified way to describe data processing pipelines (that is, generalization at the API level), but leaves open the possibility of different implementations via so-called runners (that is, specialization at the implementation level). Does this provide a way out?

Unfortunately, I don't think so. Abstractions themselves are vulnerable to the same generalization/specialization dynamics. By definition, they hide certain complexities, but over time they prove to be inadequate for their original design and invariably accumulate bloat (which is a form of specialization). My last column, for example, provides examples of how the Spark API has accumulated features over time that pollute its original elegance.[17]

In a way, abstractions actually make things worse because they enable the decoupling of the pendulum at different parts of the stack, thus allowing their motions to drift out of phase. The data processing platform might be a one-size-fits-all architecture, but the storage layer is specialized to individual use cases, and the serving platform is a monolith. And of course, legacy systems never die — today, they just get wrapped in microservices. This is one reason why modern enterprise architectures are so messy.

These observations also explain why old graybeards and curmudgeonly academics are always complaining about how the wheel gets reinvented over and over again. Because there's no free lunch, it all boils down to what tradeoffs you can live with. And that changes over time as the problem context evolves, which is why the pendulum keeps swinging back and forth. The continuous infusion of younglings means that they've never seen the cycle before, and so the same thing gets a different name each time around.

They say that those who don't learn history are doomed to repeat it. In the case of computing technology, learning history won't help much, although it will keep the graybeards from wagging their fingers at you.

Acknowledgments

I'd like to thank Tyler Akidau, Jay Kreps, Dmitriy Ryaboy, and Matei Zaharia for their helpful comments on earlier drafts of this column.

References

1. M. Stonebraker and U. Cetintemel, "'One Size Fits All': An Idea Whose Time Has Come and Gone," *Proc. IEEE 21st Int'l Conf. Data Engineering*, 2005, pp. 2–11.
2. E.F. Codd, S.B. Codd, and C.T. Salley, *Providing OLAP to User-Analysts: An IT Mandate*, E.F. Codd Assoc., 1993.
3. S. Chaudhuri and U. Dayal, "An Overview of Data Warehousing and OLAP Technology," *ACM SIGMOD Record*, vol. 26, no. 1, 1997, pp. 65–74.
4. O. Boykin et al., "Summingbird: A Framework for Integrating Batch and Online MapReduce Computations," *Proc. VLDB Endowment*, vol. 7, no. 13, 2014, pp. 1441–1451.
5. G. Lee et al., "The Unified Logging Infrastructure for Data Analytics at Twitter," *Proc. VLDB Endowment*, vol. 5, no. 12, 2012, pp. 1771–1780.
6. J. Lin and D. Ryaboy, "Scaling Big Data Mining Infrastructure: The Twitter Experience," *SIGKDD Explorations*, vol. 14, no. 2, 2012, pp. 6–19.
7. G. Mishne et al., "Fast Data in the Era of Big Data: Twitter's Real-Time Related Query Suggestion Architecture," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, 2013, pp. 1147–1157.
8. N. Marz, "How to Beat the CAP Theorem," *Thoughts from the Red Planet*, blog, 13 Oct. 2011; http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html.
9. A. Todi, "TSAR, A TimeSeries AggregatoR," blog, 27 June 2014; https://blog.twitter.com/engineering/en_us/a/2014/tsar-a-timeseries-aggregator.html.
10. J. Kreps, "Questioning the Lambda Architecture," *O'Reilly.com*, 2 July 2014; www.oreilly.com/ideas/questioning-the-lambda-architecture.
11. J. Kreps, "Introducing Kafka Streams: Stream Processing Made Simple," blog, 10 Mar. 2016; www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple.
12. M. Zaharia et al., "Discretized Streams: Fault-Tolerant Streaming Computation at Scale," *Proc. 24th ACM Symp.*

Operating Systems Principles, 2013; doi:10.1145/2517349.252273.
13. T. Akidau et al., "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing," *Proc. VLDB Endowment*, vol. 8, no. 12, 2015, pp. 1792–1803.
14. T. Akidau et al., "MillWheel: Fault-Tolerant Stream Processing at Internet Scale," *Proc. VLDB Endowment*, vol. 6, no. 12, 2013, pp. 1033–1044.
15. P. Helland, "Immutability Changes Everything," *Proc. 7th Biennial Conf. Innovative Data Systems Research*, 2015; http://cidrdb.org/cidr2015/Papers/CIDR15_Paper16.pdf.
16. J. Lin, "MapReduce is Good Enough? If All You Have Is a Hammer, Throw Away Everything That's Not a Nail!" *Big Data*, vol. 1, no. 1, 2013, pp. 28–37.
17. J. Lin, "In Defense of MapReduce," *IEEE Internet Computing*, vol. 21, no. 3, 2017, pp. 94–98.

Jimmy Lin is the David R. Cheriton Chair in the David R. Cheriton School of Computer Science at the University of Waterloo. His research interests lie at the intersection of information retrieval and natural language processing, with a particular focus on big data and large-scale distributed infrastructure for text processing. Lin has a PhD in electrical engineering and computer science from Massachusetts Institute of Technology. Contact him at jimmylin@uwaterloo.ca.