The Dataflow Model Revisited

Or: That Feeling When You Realize Every Problem You've Been Solving Is a Database Problem

Tyler Akidau
Redpanda
takidau@redpanda.com

Rafael J. Fernández-Moctezuma
Google
rfernand@google.com

Reuven Lax
Google
relax@google.com

Daniel Mills
Cambra
daniel@cambra.dev

ABSTRACT

Eleven years ago, the Dataflow Model paper argued that unbounded, out-of-order data was the new normal, and that we must stop waiting for data to ever become complete. It proposed a unified model (windowing, triggers, watermarks, and retractions) for freely trading off correctness, latency, and cost across batch and streaming engines. On the occasion of its VLDB Test of Time award, we grade our own work—a paper about streaming analytics, in truth if not in name— on what aged well, what aged badly, and what we missed.

We find the paper's core foundations largely sound: the primacy of **event time**, **the futility of waiting for completeness**, and the **insistence on strong consistency** aged well. But we got important parts of the analytical interface wrong: (1) we let windowing and triggering, whose semantics were tangled with operational concerns, dominate the exposition beyond their due, (2) triggers were an over-engineered answer to a question users should never have faced, and (3) the stream-centric worldview missed a deeper truth: streams and tables are two representations of the same object with different access semantics. The mechanisms that delivered on the paper's analytical goals ultimately evolved out of *the database playbook*: SQL, incremental view maintenance, and materialized views with explicit freshness contracts. We focused too much on the mechanics of streaming instead of finishing what the database community started but never completed: **making the complexity of analytical streaming disappear almost entirely**.

Still, the verdict is not all confession. We explore how the completeness principle split into two successful forms: **watermarks** (where streams stay visible) and **snapshot-consistent refresh** (where they do not); we trace why the latter reached far more users by asking far less of them, and generalize the former into **declared constraints on change**. We also (1) find the batch-versus-streaming debate was mostly semantic, (2) watch low-latency demand bifurcate along the old OLTP/OLAP line, leaving analytics happily at gentler freshness, (3) adopt the framing we wish we had started with (leave in, leave out, push harder), and (4) ponder the eventual disappearance of streaming beyond analytics.

PVLDB Reference Format:
Tyler Akidau, Rafael J. Fernández-Moctezuma, Reuven Lax, and Daniel Mills. The Dataflow Model Revisited. PVLDB, 19(12): 4953 - 4964, 2026.
doi:10.14778/3827998.3838710

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3838710

1 INTRODUCTION

Atul
(tossing the paper on the table)
It's not good.

Tyler
I'm sorry, what?

Atul
It's well written, but it doesn't matter. I don't want to read this. Why would I waste my time on it?

Tyler
(dies a little inside)

Thus began the reviews for the Dataflow Model paper.[1] We've all gotten blunt feedback on papers. Sometimes it hits hard, at the right time, in the right way. With elaboration, Atul Adya's tough love landed thusly: it's not always enough to have good ideas; sometimes you need to hit the reader over the head with why those ideas matter, and in language that resonates across disparate communities.

Our reaction was a manifesto aimed at everyone. The revised paper did not merely describe a model and a system; it planted a flag: *"We as a field must stop trying to groom unbounded datasets into finite pools of information that eventually become complete."* [8] That anthemic register, equal parts technical contribution and call to arms, was a key driver in the paper's eventual impact. The vocabulary it consolidated (event time versus processing time, watermarks, triggers, unaligned windows) spread well beyond the systems we built, and a decade later the paper is generously remembered as a paradigm shift in stream processing [63].

A retrospective is by definition an exercise in self-indulgence, but if you make it equal parts self-critique, it at least smells a little better in the end. The honest answer to *"did we get anything right?"* is complicated in ways we think are instructive and illuminating about the overall arc of streaming (particularly analytics) and its long and storied history with the world of databases. This paper is therefore a self-assessment, organized as follows: if we were writing the Dataflow Model paper again today, what would we leave in, what would we leave out, and what do we wish we had pushed on harder? Which of the things we built turned out to matter? And

[1] Dialogue paraphrased; memory is imperfect. But some experiences sear themselves in more strongly than others.

which of the things we worried about simply stopped mattering on their own?

One observation worth stating up front: though much of the original paper's rhetoric was framed in general terms, the Dataflow Model itself was never a truly general streaming model; it was always focused on streaming analytics. Streaming as a broad discipline extends beyond the confines of analytics, and it should be assumed that the lessons gleaned from the successes and failures of the Dataflow Model extend primarily to the realm of analytics unless otherwise indicated. We address the broader evolution of streaming across applications, infrastructure, and pipelines in Section 7. With that caveat in mind, we proceed.

Our verdict, in brief: we got the physics right, but the interface wrong. The primacy of event time, the futility of waiting for unbounded data to become complete, and the insistence that consistency was non-negotiable have all aged well (Section 2). What the field did with the completeness principle is a richer story than the one we told, and it split down two roads, only one of which the paper foresaw (Section 3). The interface is where the regrets live: windowing proved simpler than we made it out to be (just another grouping dimension), and should not have been the foundation of our consistency model. Triggers, the user-facing retraction protocol, and the stream-only worldview they were embedded in aged poorly, each an elaborate answer, in hindsight, to a question the user should never have been asked (Section 4). And the biggest miss: the database literature already held decades of the right tools (relational operators and algebras, incremental view maintenance, materialized views, and punctuation semantics), largely unrecognized or unacknowledged by us,[2] and never fully developed to their logical conclusion by the community that invented them (Section 5). We close with what we would do differently, what remains open, and where we think this line of work goes next (Sections 6–8). The aggregate lesson of the decade, and the through-line of this paper: we were all focused on the details of streaming, and although many of those details did and still do matter, the real solution for analytics was making streaming disappear almost entirely.

2 WHAT WE GOT RIGHT

For starters, the model saw broad adoption: Google Cloud Dataflow and Apache Flink built on it directly and remain under active development, and its fingerprints are on Apache Kafka Streams, Apache Spark Structured Streaming, Hazelcast Jet, Apache Samza's High Level Streams API, RisingWave, and BigQuery Continuous Queries. But adoption is a coarse measure of success. The ideas won mindshare, but did they deliver on their promises?

Three of the paper's bets aged well, essentially without caveat. Interestingly, all three apply generally across *all* of streaming, not just analytics. The things we got right, we got really right.

Event time versus processing time. The distinction between when an event happened and when a system observes it most certainly did not originate with us. The distinction runs through the time-management and out-of-order-processing literature [45, 64], but our 2015 and MillWheel [5][3] papers made it unavoidable. Per reviewer feedback, we hit the community over the head with the urgency of it, and industry responded in turn. Today it is load-bearing vocabulary in every major system [e.g., 16, 22, 55] and in the documentation of every product with a streaming story. If the paper contributed one permanent thing, that shift is it.

Consistency is non-negotiable. In 2015, streaming systems were widely assumed to be fast but wrong, an assumption institutionalized by the Lambda Architecture [46]: a weakly consistent speed layer, forever apologizing to a batch layer that produced the real answers. The paper insisted that properly built streaming systems could match batch correctness, full stop. The field agreed; indeed, the industry's own practitioners had begun calling time on Lambda well before we went to print [40]. Exactly-once processing became table stakes [26, 53, 69], and the dual-pipeline pattern has since retired to the museum of workarounds. We are also happy to report that the paper's underlying sociological claim, that users quickly stop trusting weakly consistent results, required no revision at all.

Never rely on completeness. The paper asked the field to "live and breathe under the assumption that we will never know if or when we have seen all of our data" [8]. In replacing the grooming of datasets toward a completeness that never comes with continuous adaptation as data evolves, we anticipated a world with a richer set of interesting inputs: not just append-only event logs but also changelogs of mutable state, arriving late, out of order, and subject to revision. The principle stands. Trouble was, in the same breath, we handed the field a replacement: the watermark, a completeness estimate with the model's canonical trigger built around it. So what the field ultimately *heard*, as we discuss next, was something narrower.

3 TWO ROADS FROM COMPLETENESS

If you refuse to wait for completeness, you must still decide when to say something. The 2015 paper's first answer was to reject the premise: completeness is a thing to estimate, never to await. Watermarks were the estimate, a heuristic signal of event-time progress; windows gave the estimate something to attach to, carving unbounded data into finite regions that can individually be declared done; and triggers made the pairing a language for acting before, at, and after that signal. The decade that followed took the completeness question down two very different roads, and the paper overtly anticipated only one of them.

3.1 The Stream-Centric Road: Watermarks Über Alles

Within stream-centric systems, watermarks won outright. Flink adopted them faithfully, Spark and Kafka Streams in deliberately simplified form, and the concept proved robust enough that three substantially different architectures could implement recognizably the same idea. By 2021, several of us had distilled a decade of production experience into a paper of our own, establishing formal

[2] In his defense, Rafael has always been a SQL true believer; he was the one who gave us our academic street cred in the original paper. The rest of us took longer to come around.

[3] Retrospectives also offer an opportunity to right some wrongs, and one worth highlighting here is that Tyler is often misattributed as lead author on the MillWheel paper due to the alphabetical power of his last name; Sam McVeety was the true lead author, but we made the misguidedly egalitarian decision to alphabetize everyone.

watermark semantics and comparing two of the leading implementations, Flink and Cloud Dataflow, in depth [7].

That paper also recorded a lesson the 2015 paper did not yet know, one that had taken years of production use to come into focus: not all situations *truly need* such a completeness signal [7]. The ones that do fall into two related camps:[4]

Single conclusive answers. Some computations demand one final result rather than a stream of refinements. Sometimes the action is hard to amend: the notification or alert that must fire once, correctly. Sometimes the intermediates are actively misleading: a click-through rate joined from two unsynchronized inputs will happily report values north of 1.0 until both sides converge, and its consumers prefer no answer to a nonsensical one.

Reasoning about absence. Without a completeness signal, there is no way to distinguish events that are missing from events that are merely delayed. Outer joins and anomalous dip detection are the canonical examples.

Retraction frameworks, such as in DBSP [19], can express both without any completeness signal, but only in the limit, as every result remains provisional and correctable. Yet provisional is exactly what these use cases cannot accept: a retraction can amend an answer; it cannot un-fire an alert. Revision and finality are orthogonal concerns; these use cases need the latter, and watermarks remain a practical tool for providing it.

The need for completeness is real. But the larger truth cuts the other way: many analytical applications are well served by incremental refinement alone, where an answer that updates continuously provides more utility than a static answer you have to wait for (modulo the consistency trade-offs of Section 3.2). And even the applications that demand completeness often tolerate delay: billing must be complete, but rarely within seconds; workloads sold as "low latency" rarely mean anything less. Watermarks solve completeness *at low latency*. Put together, the honest accounting is telling: needing completeness is common, and needing speed is common; needing both at once is far more rare—and yet is precisely what we designed for.

3.2 The Table-Centric Road: Ease of Use for the Win

The larger impact of the don't-wait principle arrived by a road the paper did not anticipate: materialized views. A user declares, usually in SQL, what the output should be, and the system keeps a table continuously materialized as its inputs change. The past decade filled in this road from every direction: Materialize [48], Delta Live Tables [68], Snowflake Dynamic Tables [59], Feldera [19], BigQuery Continuous Queries [34], and Flink Materialized Tables [12], all resting on the long history of materialized view maintenance [35]. The systems differ in trade-offs (append-only outputs versus generalized view maintenance, and the consistency spectrum we return to below), but they share the quality that matters: a declarative interface used to materialize a table, with the streaming left to the engine.

Tellingly, even pipelines built on stream-centric engines such as Google Cloud Dataflow and Apache Flink most often terminate in a table in a database somewhere; the destination was never in question. Those same engines have themselves been converging on declarative surfaces for describing that table—SQL of course, alongside Flink's Table API, Beam's schema-aware transforms, and its YAML API. But hand-assembling that table with a pipeline is not the same product as declaring it, and the difference is what this road is about.

In this table-centric formulation, the table represents an incrementally refined output state that usually lags the state of its inputs. This formulation is eventual consistency. In fairness to our younger selves, the 2015 paper was never opposed to it; rather, it explicitly admired the Lambda Architecture's move of offering a best low-latency estimate with correctness to follow, and set out to reproduce that pattern within a single streaming pipeline; panes (the paper's term for each successive triggered rendering of a window), refinements, and retractions were eventually consistent output in all but name. What the paper distrusted, rightly, was *weak* consistency: data lost or duplicated along the way, the *speedy* layer whose numbers users learned to stop believing. What the paper never did was name the eventual-consistency posture it had adopted and follow through on its concrete implications. The reason, in hindsight, is simple: the paper was stream-centric to its core.

Viewed through a stream-centric lens, the pitch holds together: the output is a stream of ever-better answers, and refinement is something that happens *to* the results as they flow past. But the lens is incomplete in the way that counts: a stream of refinements is, inescapably, building a table. And every question that actually matters about eventual consistency is a question about that table: What state is the result in right now? Is what I am reading internally consistent? Will it ever stop changing? The model had no vocabulary to even ask them (a gap we return to in Section 5).

Those questions deserve clear eyes, because eventual consistency does not automatically deserve trust. As we have memorably heard David Maier observe: eventual consistency is entirely compatible with perpetual *in*consistency. Precision helps here, because the term means more than one thing.

Lag. The common usage refers to output that lags input: often true of NoSQL key-value stores such as Apache Cassandra, generally true of asynchronously replicated systems, and true by definition of continuously maintained tables and views. This lag dimension is the tamer of the two. Completion information tells you which rows are complete and how far behind you are, and it already has well-worn forms: replication high watermarks in replicated systems, data watermarks or punctuations in ours.

Incoherence. The second dimension is the one that bites: internal snapshot consistency. A table should be self-consistent at every moment, even while it lags its inputs; if a query references the same rows from two subqueries, both should agree on the set of rows processed. Obvious as that sounds, streaming pipelines have routinely failed to provide it.

The 2015 paper's implicit answer was windowing: delay materialization until the window closes, and consistency follows. But a user should not have to modify their query's semantics, adding a grouping they never wanted, to obtain a guarantee the system should have provided. Windowing as a consistency crutch can even break the query outright: add a tumbling window to a join, and

[4] A third concern belongs to the engine: *obsolescence*, knowing when buffered state can safely be reclaimed. This one is internal machinery, not user semantics.

rows falling on opposite sides of a window boundary silently stop joining. A windowed aggregation belongs in a query to produce windowed results, full stop.

Fortunately we see systems now being built that provide snapshot isolation or stronger [3]. The distinction from mere freshness is worth spelling out. Under snapshot isolation, a result is still only eventually consistent with respect to the final answer, but it is *perpetually consistent* as a view of the world: at any given moment, everything you read reflects one coherent state of the inputs. This holistic coherence is what makes the easy, table-centric interface safe for the vast class of correlation-shaped use cases (joining and comparing across sources) that quietly fall apart when each table reflects a different moment. Lacking it, eventual consistency decays into Maier's perpetual chaos.

How much coherence, spanning how many tables, is where systems differ. Per-view atomicity is the floor, and increasingly table stakes. Delta Live Tables extends refresh consistency across a pipeline [68]. Snowflake Dynamic Tables discards the pipeline boundary entirely: refresh timestamps are guaranteed to align across any set of tables in the account, regardless of differing target lags, but reads are consistent only within a single table (product decision, not architectural limit) [59].

Materialize and Feldera hold the strongest line: every read, at any moment, observes all views at a single timestamp [19, 47]. A coherent view across *everything you read at once* remains the differentiator, not the floor; coherence across everything you *maintain*, meanwhile, has quietly become the expectation.

This formulation delivers everything analytical users wanted from streaming, minus everything that made streaming hard. The road rests on classical incremental view maintenance, modernized by Timely and Differential Dataflow [49, 52], change queries [6], DBSP [19], and Enzyme [68]. All of this work did much to put the 2015 paper's stated goals (correctness, latency, and cost, tunable per use case) into the hands of ordinary users uninterested in authoring or maintaining streaming pipelines. It achieved this chiefly by never asking users to think about streams at all.

3.3 The Road Not Taken: Declared Constraints on Change

Both roads were grasping toward the same underlying concept: completeness. Our model gave it exactly one embodiment, windows finalized by watermarks. But as Section 3.2 argued, windowing is an aggregation dimension, not a consistency mechanism, and the watermark is only one kind of completion signal.[5] Completeness in tables generalizes along two axes.

Generalizing the first axis: what window finalization really asserts is that a region of data has become *immutable*. There is no reason the boundaries of that region must be monotonically advancing points in event time; it can be an arbitrary declared predicate, provided the immutable region never shrinks. The predicate may be defined in terms of a watermark or punctuation, but business logic is free to dictate others. For example, an account being closed may mean that all associated rows are now immutable.

[5]That windowing *delivered* completeness and consistency was never the problem. The quibble is that the model pitched it as the *only* way of obtaining them.

The industry has been converging on this generalization (*finalization*) from more than one direction. Delta Live Tables made early strides in this space: its streaming tables treat append-only input as processed-once and never revisited, with `APPLY CHANGES` layering change data capture on top [27]. A more imperative approach, but effective nonetheless. Four years later Snowflake delivered frozen regions on Dynamic Tables [58]: a declarative predicate that marks rows immutable, allowing engines to skip re-verification of frozen data, downstream consumers to treat it as final, and backfilled history to remain stable even when upstream sources diverge.

Generalizing the second axis: there is no reason progress must be measured *only* in time, or even in time at all. The ladder of generality was visible all along, as several of us later surveyed in detail in the watermarks paper [7]. Watermarks track a single time domain. Timely Dataflow's timestamp frontiers [52] let multiple time domains advance independently and are the machinery beneath Differential Dataflow's simultaneous incremental-and-iterative processing. Punctuations [64] subsume them all: completeness signaled for arbitrary predicates over the data, a canonical notion of progress acknowledged in both our 2015 Dataflow and 2013 MillWheel papers [5, 8] as long anticipating our notion of watermark. The lineage keeps resurfacing: sequence-based progress tokens in modern table systems, and most recently *watches* in the Pub/Sub setting [2], proposed in a symmetry we most certainly did not engineer, by two reviewers acknowledged in our 2015 paper.

It is also no accident that the NEXMark queries [65] remain the field's benchmark staple: they were written by people who understood progress as something more general than a clock. So why did the doubly special case win? Because generality, in those formulations, billed by the operator: punctuations require every operator to correctly update and propagate arbitrary predicates, frontiers make sophisticated operators intimidating to write, and most use cases need neither. Watermarks were a deliberate sweet-spot choice, not an oversight [7].

Put the two generalizations together and a vocabulary emerges that we think is the right way to have framed the whole space: *declared constraints on change*. What can no longer change (finalization); in what order changes may arrive (ordering); in which direction values may move (monotonicity); how change is scoped (partitioning). Time still appears, but only as one dimension a constraint might reference, not as the substrate of the vocabulary itself; the watermark does not necessarily go away, but a table is free to decide whether to use it for finalization. A constraint can still reference watermark-based window finalization, but is no longer limited to only that approach. These are properties engines can verify and exploit. Crucially, they are also easy for the engine to manage and propagate. Were we to write the paper again, this would be our central completeness narrative.

4 WHAT WE GOT WRONG

4.1 Windowing and Triggering, Entangled

The 2015 paper's most elaborate machinery is the windowing formalism. With concepts such as assignment and merging as separable operations, and sessions as the motivating case, the formalism was novel in its merging half, while assignment we borrowed

openly from Li et al. [44].[6] Even on its own terms the formalism was incomplete: merging has an inverse, *splitting*, and we missed it; windows in our model could grow but never shrink.[7] Temporal validity windows—in which each value in a relation is valid until superseded, so that a late-arriving successor quietly truncates the reach of its predecessor, splitting whatever data the shrunken window once contained—require exactly the operation we never provided. The use case is as practical as they come: it is how you join against a table of currency-conversion rates in event time [36]. One of us later spent a book chapter working around its absence [9], and the temporal-database literature had, naturally, been modeling validity intervals for decades: valid time dates to the field's founding taxonomy [56], and had reached the SQL standard [41] years before our model failed to express a window that shrinks.

The trigger language, meanwhile, was expressive enough to reconstruct every emission pattern we had ever encountered: composites, sequences, repetitions, data-driven firings, custom signals. Yet almost nobody used either windowing or triggers at anything close to full generality. Said differently, it was a nightmare of unnecessary complexity. A decade of production distilled the trigger menagerie to two members: fire when the watermark passes, and fire periodically. The windowing taxonomy, whose full depths later required a survey of its own to catalog [67], collapsed for most users into "group by a time bucket."

The *clunkiness* of triggers was more than cosmetic; it was structural. Triggers attached wherever windowing was declared, deep in the interior of the plan, and propagated outward from there. When a grouping lived inside a library transform there was frequently no sane place to specify its triggering at all. The design also smuggled in an obligation the model never honored: a sequence of triggered refinements is only meaningful if consumed in order, final pane last. This obligation is really an ordering requirement we return to in the next subsection.

The fix, from the model's perspective, is to invert the arrangement entirely: the purpose of triggering is to control emit frequencies, so the emit policy belongs at the root of the plan (the sink) and should be pushed *down* by the engine toward whichever interior groupings require triggering, precisely the way database optimizers have pushed predicates down their plans for decades. That inversion matches what users actually intend, dissolves the ordering obligation into the engine where it belongs, and maps naturally onto SQL, where nobody can set a trigger on every grouping, but everyone already attaches export policies at the outputs (`EXPORT DATA`, `COPY INTO`). Stated as a design principle: separate the semantics of progress from the mechanics of processing, and let the schedule of result production be declared from outside the query rather than programmed within it.

Production systems have since validated the inversion, and further generalized it. Delta Live Tables validated it first, with declared refresh schedules at the scope of a pipeline [28]. Snowflake then generalized the contract: a Dynamic Table's target lag may be declared at any node in the dependency graph, not merely at the outputs [57, 59]. Interior tables that exist purely as plumbing declare their lag as `DOWNSTREAM`, deriving their cadence from whichever tables consume them (precisely the sink-driven derivation just described), while the scheduler reconciles competing contracts by honoring the strictest downstream demand and refreshing the graph in dependency order. The generalization is the table-centric worldview asserting itself once more: in a pipeline, only the outputs speak for freshness; in a database, every node is potentially somebody's output, and so every node must be allowed to speak.

Meanwhile, BigQuery moved the contract across the divide entirely. Classic materialized views, such as in BigQuery, Snowflake,[8] and many other databases, had always held the read side at its extreme: results current as of the query, whether by transparently merging the view with base-table changes at read time or by maintaining the view synchronously with each write, at whatever cost that freshness implies. `max_staleness` made that guarantee negotiable [33]: within the declared bound, a query is served from the view as-is; beyond it, the merge-on-read kicks in—freshness as a contract the reader declares and the engine prices, rather than a fixed property of the view.

The diagnosis, we think, is not that the machinery was wrong in its details; it is that both pieces were promoted far above their station. Windowing is a time-flavored `GROUP BY`, "just a slight modification of a thing everyone already innately understands: grouping" [9]. In other words, one grouping construct among many, not the center of a model. And what users wanted from triggers was never a language for *when*; it was a contract for *how fresh*. The construct that survived contact with real users is that of *target lag*, the emission contract in its purest form: a single declared bound on staleness, from which the engine derives every scheduling decision the trigger language would have made the user specify. The engine can optimize against this construct, batching and amortizing work in ways an imperative firing rule forbids. The construct now ships across the industry under different names, such as `refresh triggers` in Delta Live Tables (2021), `max_staleness` in BigQuery (2022), `TARGET_LAG` in Snowflake (2023), and `FRESHNESS` on Flink's Materialized Tables (2024) [12]. One idea, four spellings. Declarative freshness turned out to be to triggers what the relational model was to CODASYL's navigational databases: once users could declare the result they wanted, nobody missed telling the system how to get there.

The expressive benefits of a declarative approach were not unknown to us at the time: one of us had previously explored letting consumers request results from a continuous query via punctuations flowing *against* the stream, expressing both the intent to produce (or suppress) results and the subset of the stream referenced, in terms of the output schema [31].

The general lesson is the one this paper keeps returning to: operating on the stream is often too hard for mere mortals. Users should declare the result they want. Deriving the incremental computation—the actual streaming—is the engine's job, and the province of experts who enjoy that sort of thing (e.g., [6, 19, 49, 52, 68]).

[6] And while we are redistributing windowing credit, an erratum: the 2015 paper stated the windowing semantics of CEDR [17] and Trill [23] were insufficient to express sessions. CEDR could express them via left anti-semijoins, leveraging the temporal construct of the data model.

[7] We don't claim assign-plus-merge-plus-split completes the algebra; it merely patches the hole we happened to fall into.

[8] Not to be confused with Dynamic Tables.

4.2 The Ordering We Never Promised

The model declared every collection unordered. The choice came from two sides: functional programming and an obsession with scale-out parallelism. This choice was strictly speaking a falsehood no one could realistically live by. Real pipelines routinely require at least weak ordering: any upsert-shaped output where the last write wins is implicitly ordered, and the model's own triggering semantics—early panes followed by an on-time pane, refinements followed by retractions—are meaningless unless those outputs are observed in order. The saving grace, also unnamed in the model, is that global order is almost never the actual requirement: an upsert stream into a keyed store needs only per-key ordering, and sinks today recover even that ad hoc, with sequence numbers here and last-writer-wins there. In practice, users leaned on ordering guarantees that specific engines happened to provide and the model never promised. Our miss: we pitched the model without getting into sufficient detail to formulate ordering semantics, even though we were painfully aware of them in practice. It is one more exhibit for the previous subsection's lesson: stream-level contracts are treacherous even for the people who write them.

4.3 The Latency Realities We Left Unstated

What did the 2015 paper actually claim about latency? Less than its reputation might suggest. It said that one can "never fully optimize along all dimensions of correctness, latency, and cost," and that systems must provide tools for balancing the three "appropriate for the specific use case at hand" [8]. That text has aged shockingly well; it applies generally across streaming even today, and in analytics, a declared target lag is its thesis shipped as product. We were not even living at the low end of the dial ourselves: our operational concerns ran closer to single-digit seconds than single-digit milliseconds. On its face, there is nothing here to confess. But this subsection earns its place among the regrets twice over.

First, the paper oversold the dial. It promised "the ability to dial in precisely the amount of latency and correctness for any specific problem domain" [8]; physics declines to honor that promise at the dial's low-latency end. When data volumes are high, state-of-the-art systems can muster seconds of latency at high percentiles for their target use cases, but we do not expect generalized large-data systems to reach milliseconds: the optimizations that make queries efficient (batching for vectorized execution among them) evaporate at very low latency, and taming environmental noise such as noisy neighbors means under-utilized machines and more expensive queries.

Nor is even gentle freshness universal. A `MIN` aggregation under modifications and deletions requires a full rescan to maintain incrementally; the Reactive Aggregator [61] lowers that to $O(\log |\text{table}|)$, still impractical every ten seconds on a large table; and the approximate variants that *can* keep up surrender exactness to do it. Operators resist incrementalization in ways rarely obvious to users, so the system must make the trade-offs on their behalf.

Importantly, notice what this example embodies: the paper's first quote above, made concrete. Pin latency at scale and cost or correctness must give. "Never fully optimize along all dimensions" was the truer sentence; the dial promised a freedom the paper's own caveat had already revoked.

Second, the paper never said where along the dial the demand actually lives, and it was written from the vantage point of an industry that had already decided the answer was "as low as possible." But as we found in the years that followed, everyone needs low latency until you tell them how much it costs... then for a surprising number of use cases, that requirement just melts away. The demand, it turns out, bifurcates, and along a line databases had already drawn decades earlier: OLTP versus OLAP.

Operational and transactional uses of streaming (the applications and infrastructure we take up in Section 7) skew genuinely toward millisecond latency, by definition: an application responding to the world must keep pace with it, often within human reaction time. Analytics, the 2015 paper's real subject, returned the opposite verdict: seconds-to-minutes freshness, delivered cheaply, reliably, and without operational heroics, serves the wide middle of analytic use cases—and a long tail is content with far less. The systems that met users there flourished. Ultra-low-latency analytics remains vital for a narrow tier of workloads, and genuinely impressive engineering serves that tier, but it is a tier, not the market. Nor was the engineering wasted: the incremental machinery built in pursuit of milliseconds is much of what now serves the broader market at gentler freshness.

Still, much of the industry (and few more enthusiastically than us) spent years optimizing the corner of the analytical latency–cost–correctness space with the fewest customers in it. A paper about balancing correctness, latency, and cost was the natural place to say where the customers were, and it stayed silent. The framing was not wrong; our collective aim was, and the text never tried to correct it. Once again, what the field took away was not what the paper said.

4.4 The Peace Nobody Heard

On batch versus streaming, we will partially defend our younger selves: the claim was right. The 2015 paper declared that execution engines should not dictate semantics, that batch, micro-batch, and streaming systems could all provide equal correctness, and that engine choice should reduce to "the practical underlying differences between them: those of latency and resource cost" [8]. A decade later, that is simply how the winning systems work: streaming semantics computed on whatever machinery the freshness contract will pay for. Change queries over table snapshots [6] are streaming with batch technology; a declared view refreshing every minute is micro-batch wearing a declarative suit.

What we got wrong was the articulation, and the articulation mattered. Since the word "streaming" means two different things, the war fed on the ambiguity. There is streaming the *semantics*: computation over data that changes over time, the conceptual model that strictly subsumes batch (a bounded dataset is a stream that happens to end), and there is streaming the *engine*: continuous, record-at-a-time[9] execution, which is merely one point in a space of execution strategies (batch, micro-batch, true streaming) with genuine strengths and materially different trade-offs (a Flink is not a Spark, in both directions). Batch as an engine is superb technology:

[9]Nominally. No production engine is truly record-at-a-time; there is always batching under the covers. The important questions are how much, where, and what limitations are imposed as a result.

efficient, throughput-optimized, operationally simple, and very hard to beat on cost. The 2015 paper saw the distinction; it even tried to legislate it, reserving "batch" and "streaming" exclusively for execution engines and pushing "bounded" and "unbounded" for data [8]. The field kept half the memo. Bounded and unbounded were then in the vocabulary, but "streaming" swallowed the concept anyway and inevitably some discussions over a decade have gone to defending engines as though the semantics were at stake. With two words for the two things, the peace is self-evident: adopt the semantics everywhere, and choose the engine by the freshness contract. With one word for both, we got an engine war our own claim said was unnecessary.

The war ended only when the categories themselves went away: in the table-centric systems, whether a given refresh executes as a full recomputation or an incremental delta is the engine's decision, made per table and per moment, invisible from above. One sign of the peace is that Apache Flink itself now ships it: a Materialized Table declares its `FRESHNESS`, and the engine converts that one number into either a continuous streaming job or a scheduled batch refresh [12]. When viewed from this angle, there is nothing left to fight about and no need for a treaty; the war's subject simply dissolves.

5 WHAT WE MISSED

5.1 Streams and Tables

The most consequential omission is easy to state: tables. The word barely appears in the 2015 paper, yet tables are everywhere in it, unnamed. `GroupByKeyAndWindow` takes a stream and produces evolving keyed state (a table). A trigger observes evolving state and emits its changes (a stream). The paper's figures drew both halves of this duality repeatedly; its authors missed the nouns. In the candid words of the lead author: "I just didn't fully understand what I was talking about yet. Streams and tables were an epiphany when I finally understood them."

In fairness, the database playbook already had entries moving in that direction: CQL's conversion operators shuttled data between streams and relations [15], and TelegraphCQ ran continuous queries over both side by side [25]. But the duality itself went unstated (streams and tables remained two things to convert between, not one object seen two ways),[10] and the complex packaging did the ideas no favors. So despite knowledge of these works, the epiphany awaited.

The vocabulary that finally took hold came, true to this paper's thesis, from the database world, and it was already in the air as we wrote: as popularized by Kreps, Kleppmann, and the community around Apache Kafka [38, 39, 55], the duality had been hiding in the architecture of every database all along. The raw materials were never secret: recovery had long treated the log as the ground truth from which tables are rebuilt [51], warehouses had shipped deltas between systems for years as change data capture [42], and every materialized view demonstrated the round trip; it was all simply filed under mechanism, not model. Read together, the mechanisms spell out the model: a stream is an ordered sequence of changes (what happened), a table is a stream of snapshots (what was true when), and each is recoverable from the other [36].[11]

One of us later spent a book chapter exhuming the tables from the Dataflow Model along exactly these lines [9]: every operation in the model classifies as stream→stream (element-wise), stream→table (grouping), or table→stream (ungrouping, which is all a trigger ever was), and even a MapReduce job comes apart into streams and tables from top to bottom. Then came the formalization: the time-varying relation [18], of which a table is a point-in-time view and a stream is the changelog. One object, two representations, with the entire relational algebra remaining meaningful over it. Meijer had even explained the same duality for .NET's enumerable and observable collections years earlier [50].

Had we understood this in 2015, most of the paper's machinery would have simplified: retractions stop being an exotic protocol and become ordinary changelog rows; triggers stop being a firing language and become a materialization policy; and the "unified model" we claimed arrives not by bridging batch and streaming APIs but by recognizing they are different views of the same thing.

Yet even once we had the vocabulary, we still did not internalize all of its consequences. Consider the mechanisms for in-place evolution of a running stateful query, which Apache Flink and Google Cloud Dataflow both model as an out-of-band operation: upgrade from a savepoint in Flink, job replacement in Dataflow. Viewed through the duality, an update is just more stream: a structural-change marker in the changelog, which is how many databases already record schema modifications. The machinery sat outside the model only because nobody asked the model to describe it.

5.2 SQL

We bet on a high-level programming model in a general-purpose language, largely ignoring SQL. The value of SQL here is threefold and we underestimated all three: *reach*, because every analyst on earth can write it and none of them will learn a streaming API; *declarativeness*, because a query the user did not procedurally specify is a query the engine is free to incrementalize, optimize, and re-plan, hence enabling the entire database optimization toolkit to become freely available; and its quietly *time-varying nature*, because a SQL query over time-varying relations is already meaningful as both a point-in-time question and a continuously maintained one. SQL never had a batch/streaming split to heal. Almost.

Table-native SQL did need one genuinely new primitive: a way to extract the stream from the table, specifically, the changes between two states of a time-varying relation, whether surfaced as a `CHANGES` construct [6, 32] or an `EMIT STREAM` clause [18] (CQL's `Istream` and `Dstream` had come tantalizingly close two decades earlier, outside the standard [15], but as two streams to juggle rather than one changelog). In the narrow accounting of language surface, the standard already held nearly all of the answers, and the streaming decade's contribution was a single missing primitive: the reverse of the ratio we would have guessed in 2015. But that accounting is narrow indeed, and we return to the fuller ledger below.

[10] CQL came the closest: `Istream` and `Dstream` are true change streams out of a relation. But the only road back was a window (excerpts, not integration), so the round trip never closed.

[11] Hyde's rendering of the duality is the most physical: a table's value over time is the Heaviside step function, and its stream of changes is the Dirac delta, the derivative of the step. Replaying a log and snapshotting state are the two directions of the fundamental theorem of calculus, wearing database clothes.

Ignoring SQL completely was a mistake; claiming SQL for everything would be the opposite mistake. Expressiveness is not actually the issue (recursive common table expressions make SQL Turing-complete, so any computation can in principle be written), ergonomics is: an important population of users prefers a general-purpose language because some logic is genuinely awkward to state declaratively, and closing that gap is a challenge for the SQL community to take up or decline.

Whichever surface one prefers, though, our data model missed another opportunity. The 2015 model made no assumptions about the structure of elements: to the model they were opaque values, their interpretation left entirely to user code (in Apache Beam's embodiment, user-registered `Coders` decoding blobs into whatever types the pipeline author fancied). A decade of experience says nearly all real-world data fits a relational schema, and the sliver that does not (encoded video, say) fits a single binary column. Building on schemas and a relational algebra from the start would have bought a semantic layer that users could reason about and engines could plan and optimize. This is a lesson demonstrated since by Beam's own schema API [11] and Flink's Table API [14]. Blobs gained us generality precisely where nobody needed it, at the cost of everything an optimizer could have done with the structure we threw away.

5.3 The Answer Neither Community Finished Alone

Incremental view maintenance is the user-friendly answer to streaming analytics, and is thirty-something years old [35]. This is the keystone "That Feeling When" of our title, and it deserves to be stated plainly: the problem the 2015 paper attacked with windows, triggers, watermarks, and retractions (keep a derived result continuously correct as its inputs change) is the materialized view maintenance problem, posed and substantially theorized by the database community while most of us were still in school.

But the title's feeling cuts both ways, and here is the second edge: if every problem we worked on was a database problem, these problems belonged to the database community all along, and the community that invented the answers did not finish them either. Classical materialized views arrived with restrictive query support, opaque staleness, and ergonomics that assumed a DBA in the loop; they remained a niche feature for decades, and while the theory kept advancing (higher-order view maintenance [4] would eventually feed DBSP [19]), the database world largely stopped pushing the product.

Getting from that foundation to systems ordinary users adopt took the streaming decade's contributions. The theory hardened into general incremental-update frameworks: first Timely and Differential Dataflow [49, 52], then DBSP [19], whose elegant reduction turns the incrementalization of any query into a mechanical, provably correct circuit transformation. Onto that base came event-time semantics grafted where they matter [18], change derivation over arbitrary query plans [6], and above all the ease-of-use discipline of a declared query plus a declared freshness bound, with everything else automated [33, 59, 68]. Nobody was fully right. They had the answer in their grasp and did not see it; we reinvented fragments of it under new names and repeatedly failed to make the connection. It took both communities, and more than three decades, to finish the thought.

It is perhaps no coincidence that the database half of this story was written largely in research venues and the streaming half largely in production systems. The two halves complemented each other in ways neither could have planned. The research community worked out the principles at a time when the workloads that would demand them at planetary scale did not yet exist: theory ready and waiting decades ahead of its demand curve. The practitioners, when those workloads arrived, chased them with more urgency than scholarship, and kept rediscovering principles whose proper names we learned only later. Our title concedes the concepts, and it should: the concepts were theirs. But it is only half serious, because priority is only half the story; finishing the thought took both the ideas and their delivery. And if the streaming decade earned anything in this telling, we hope it is a seat at the table it took us a decade to see.

Regardless, the result was worth the time and effort it took on both sides. A user declares a table (any SQL query) and a freshness contract. The engine streams. The user never sees it. That is what the 2015 paper was reaching for, and it is instructive that reaching it required deleting nearly everything the paper spent its pages on from the analytical user surface.

There is a final irony here, best stated in the vocabulary of the Beam Model's take on stream-and-table theory itself, summarized in Figure 1. The operation taxonomy in that theory has a deliberate hole: there is no table→table operation, because data cannot pass from rest back to rest without moving in between [9]. Yet a declared table maintained over other tables is precisely a table→table operation at the interface. The physics have not changed; the streams are all still in there, doing the moving. They have simply been internalized by the engine. The one operation our own theory said could not exist turned out to be the product everyone wanted, and manufacturing the illusion of it is, quite literally, what it means to make streaming disappear.

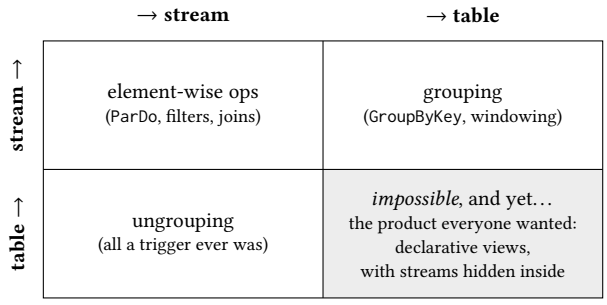

Figure 1: The stream/table operation taxonomy [9]. The fourth cell cannot physically exist as data cannot pass from rest to rest without moving in between. The winning interface manufactures the illusion of it: the engine does the moving, invisibly, making streaming disappear, as a picture.

6 IF WE COULD TURN BACK TIME

If we could redo the work all over again with what we know now, we would leave some things in, remove others, and maybe relax a bit regarding a few things that otherwise resolved themselves.

6.1 Leave In

Everything in Section 2: event time, consistency-or-bust, the completeness principle. We would also keep unaligned windows. Sessions were and remain a genuinely important use case, and merge-based window assignment proved the right mechanism for them. However, we would keep them on the terms of Section 4: as grouping semantics an engine implements (complete with the splitting operation we forgot), not as a centerpiece users program against. And we would keep the ambition, if not the framing, of engine-independence: the claim that semantics must not be dictated by the execution engine was correct and foundational, even in light of the overlooked connection that relational models had already anticipated this idea.

6.2 Leave Out

Most of the trigger language, for the reasons of Section 4. And *user-facing* retractions, with heavy emphasis on the qualifier. The mechanism itself turned out to be triumphantly correct. In the systems descended from the 2015 model, retractions as a user-visible protocol were barely implemented and almost never used as designed. We were not even the first to try: Borealis had made revision messages (insertions, deletions, replacements) first-class citizens of its data model a decade earlier [1], complete with the proliferation problems and opt-out applications that foreshadowed our own protocol's fate.

But hidden beneath the declarative surface of the view-maintenance world, retractions are not merely used; they are critical. Incremental view maintenance *is* the propagation of deltas and their negations through a plan. We find this mechanism in the diffs of Differential Dataflow [49], the retract streams Flink's SQL planner threads between operators [14], change queries in Snowflake [6], the deltas of Delta Live Tables' Enzyme engine [68], and the Z-sets of DBSP [19]. The idea was entirely right. What was wrong was the audience: we handed an engine's mechanism to users and asked them to reason about it in the raw. When consistency is the engine's job, the user never needs to see a retraction at all.

The one place retractions rightly stay user-facing is the stream boundary itself: when the stream is the product, as in change data capture, deletes are retractions deliberately surfaced, and nobody asks their CDC feed to hide them. In table form, retractions are the engine's business; in stream form, they are the contract. It is the paper's pattern in miniature: right mechanism, wrong altitude.

6.3 The Worry That Became Commodity

We worried a great deal in 2015 about pipelines owning their fault tolerance. Strongly consistent state, exactly-once processing, and checkpointing protocols were simultaneously headline features, research topics, and competitive differentiators for every engine [5, 22]. The core worry remains legitimate: state must survive failure. Recomputable inputs don't change this math: recomputing a year of history from a raw log because a node died is economically absurd, which is why serious systems still persist intermediate state.

What changed is that this machinery became commodity, in both senses of the word: implemented everywhere, and assembled from parts everything else already shared. Popular stream-centric systems accomplish this whether via fine-grained checkpoint protocols in MillWheel [5] and its successor Google Cloud Dataflow [43], deterministic micro-batch replay in Spark Structured Streaming [16, 69], or Chandy–Lamport snapshots in Apache Flink [22]. For databases, durable logs provided a floor to rebuild from, while transactional table storage gave intermediate state a home that was already snapshot-isolated and durable. In the view-maintenance world, intermediate state is tables all the way down, and recovery is just the next refresh reading committed snapshots. Fault tolerance stopped being a bespoke protocol each engine had to invent and became one more thing from the database toolbox, so it, too, disappeared.

Notably, this disappearance should be scored as a victory, not an anticlimax. Exactly-once was only ever a boast because the industry had spent years decommoditizing correctness: no 1980s database would have dreamed of advertising that its aggregations didn't miscount. That we had to say it out loud for so long was the anomaly. That nobody needs to say it anymore is the ship righted.

6.4 What We Wish We Had Pushed

Remarkably little, it turns out. We expected to fill this bucket generously, but found our regrets were of other kinds: mechanisms we pushed too hard (Section 4) and truths we failed to see (Section 5). What fills it instead is *demand*: new expectations the world has since placed on the model, each deserving an honest answer rather than a retroactive wish. There are two, and under this paper's scope they turn out to share an answer: both are, chiefly, demands from beyond analytics.

First, the model committed to DAGs and did not include iteration. Back edges were awkward guests in the formalism, so we left them out. The clearest demand since comes from workflow automation, agentic systems most recently: stateful, event-driven, long-running programs that react to the world and feed results back into themselves. That feedback loop is a stream-processing workload with precisely the back edge we never modeled, and serving it with the model today means simulating cycles through external message queues. So, honestly: we missed cycles.

But cycles are complex for analytics on both sides of the interface: the engine needs richer progress tracking than singleton watermarks and new semantics for state lifetime, and the author needs to reason recursively, something the average business analyst does not reach for on a whim. The analytic appetite is correspondingly real but modest (SQL's recursive common table expressions, Differential Dataflow's iterative computations [49]), and the use cases that justify the complexity skew heavily operational. That demand belongs to the story of Section 7, and while meeting it will take real invention, tracking progress through cycles will not: the flying fixed-point operator was doing so for cyclic stream queries in 2009 [24], and Timely Dataflow's frontiers solved it again in 2013 [52]. That much of the territory is charted.

Second is a primitive regularly nominated for our "should have talked about it" list, one we possessed from the start but declined to push: state and timers. MillWheel was state and timers everywhere—manually managed per-key state, explicitly set timers—and the 2015 model was largely an attempt to abstract that machinery away. The primitive was implemented when we wrote the paper, but we lacked consensus to expose it. Looking back, we remain only half repentant.

The repentant half: in analytics, the most common use of state and watermark-triggered timers was to access the completion predicate. This goes back to our earlier premise that windows were promoted above their station: many use cases needed to know when data was complete but did not fit cleanly inside the windowing model, so users manually buffered state and relied on watermark timers to learn when it was safe to act. The correct answer is the one from Section 3.3: windowing is for aggregation, not consistency, and generalized finalization predicates are the correct way to expose completeness semantics in the model.

The unrepentant half: most of what state and timers serve is operational rather than analytic. The model later incorporated a public state and timers API, which has proven enduringly popular; Flink offers a near-identical API and has built an entire framework atop it [13]. Look at what those pipelines do, and two jobs dominate: large-scale state machines that are not analytic queries at all, and sinks into other systems.[12] Neither job is necessarily beyond declarative analytics (SQL's `MATCH_RECOGNIZE`, another type of partitioned state machine, comes to mind), but we still feel these constructs are an escape hatch—the assembly language of streaming—rather than a core part of the analytically minded Dataflow Model; tellingly, the view-maintenance generation that delivered on the model's analytical goals has so far felt no need for an equivalent.

The market, for its part, has since confirmed the diagnosis. As the MillWheel era's workloads dispersed onto newer generations of systems, analytics settled into view maintenance, while stateful workflows landed in purpose-built homes of their own: the durable-execution lineage running from AWS's Simple Workflow Service [10] through Uber's Cadence [66] to Temporal [62], joined from the database side by DBOS [29] and from the streaming side by Restate [54]. Notably, those systems kept state and timers, wrapped up in friendlier trappings. What amounted to an escape hatch in analytics is a foundational construct in workflow automation, and this is where the abstractions have been rising to meet the use case.

7 THE LARGER DISAPPEARANCE

We have argued extensively that streaming analytics is finding a happy ending as its complexity disappears into the database, but we dare not imply one size fits all [60]. The honest coda is that it disappeared *only* there. Within analytics, the taming is real: gone are the days of a small army of developers crafting streaming pipelines, and an expert data analyst now gets their work done with the same ease as any other interaction with the database. But this claim falls short when applied to streaming generally.

Step outside of analytics and into applications, microservices, and APIs, and streaming's complexity reappears at once: queues, callbacks, retries, dual writes, hand-plumbed consistency between systems that share no model at all. The reason is structural: each component has a coherent internal model, but components interact at a lower level (bytes, addresses, packets, connections). The overall system is only as coherent as the model its parts share, yet the modern stack is a mess of incompatible models. Streaming analytics disappeared into the relational database because the relational model is expressive enough to cover its whole domain. For the entirety of streaming to disappear, and not just analytics, it needs a sufficiently general model to disappear into.

What the database supplied was a *collapse*: the changelog and the point-in-time relation, recognized as one object, freeing the engine to choose the physics beneath a fixed meaning. Nothing about that requires the object to be a relation. A program with a declarative meaning and a streaming-dataflow operational behavior is the same duality over arbitrary computation: what it *means* is fixed, how it *runs* is the engine's to decide.

Realizing this requires generalizing along two dimensions: (1) *the result*: the language must express arbitrary application logic, not just relational queries over tables; and (2) *the operations*: the single variable of freshness must expand into a richer envelope of latency, durability, priority, and cost.

The payoff is verification and branching: properties an engine can establish rather than test at the seams, and system state that is a value (snapshot-isolated, forkable, replayable). Nor do AI agents make any of this moot: abstraction is what keeps complexity tractable as a system grows, and the plumbing agents hand-build today is exactly the fragmentation a coherent model erases.

This is the direction the work of one of us now points [20, 21]; whether it can be pulled off beyond analytics, we do not yet know. But the transformation that overtook analytics looks ready to happen wherever software is still assembled by hand from mismatched parts. We are curious to see where things stand in another eleven years.

8 CONCLUSIONS

Did we get *anything* right? Yes: the physics. Event time, the refusal to wait for completeness, and the insistence on consistency were correct. These concepts were worth planting a flag over, and the field is better for having adopted them. No: the interface. Windows as a consistency mechanism, triggers, retractions, and the stream itself were the wrong things to hand a user. Nearly everything the paper elaborated most lovingly has been abstracted away by systems that ask for a query and a freshness bound. The part we can neither fully claim nor entirely concede: the destination was in the database literature all along, invented decades earlier, left materially unfinished by its inventors, materially underappreciated by us, and completed at last by both communities together.

One final confession, long owed and long suffered: "Dataflow Model" was an unfortunate name, borrowed from the product. We were well aware the name collided with not one but two generations of prior art on the word: Kildall's dataflow analysis [37] and DeMarco's dataflow diagrams [30], among others.

On another note, had this paper come a little later, it would have ended up the "Beam Model" instead, which is how the community that still fosters the original model now refers to it. No more

[12] In the database world, sinks are typically provided as built-ins; here, users appreciated being able to build their own rather than wait for someone else.

illuminating, far less hackle-raising. While preparing this retrospective, we admit, eleven years later we still don't have a good product-agnostic name for the model.

"That Feeling When," indeed. In the database community, that phrase is a running joke, but as many of you know, there is nonetheless a certain ache to discovering your hard-earned "invention" is the punchline. With time, however, the ache mellows into respect and gratitude, and we could not be happier to see these ideas at work in so many places, regardless of their provenance.

May the next decade's builders read from as many communities as possible. May their reviewers be as merciless as ours. And may they feel the joy and pride of seeing their creation adopted as broadly as ours was, minus the discovery that something simpler was there all along.

ACKNOWLEDGMENTS

Our deepest thanks to Atul Adya, for the 2015 review that gave the paper its heart and for embodying, entirely unwittingly, this paper's thesis: that the database community had the foundational answers all along, and we failed to recognize them even when they were standing right next to us. (Tyler literally knew Atul for 16 years before finally making the connection he was *that* transaction isolation guy. The shame!) To Colin Meek for lending his deep and thoughtful review to the more technical corners of the original paper, and for his early feedback on this retrospective. To David Maier, on whose work we stood more than we realized, and whose insights sharpened this piece. To Saurabh Bansal, Kory Kraft, Yaroslav Litus, and Bill Neubauer for their observations and suggestions. To Dan Sotolongo, fellow traveler from triggers to tables, now bearing the torch with Daniel toward One Model to Rule Them All, for his gracious feedback on this paper. To Grzegorz Czajkowski, who backed the original work when it mattered most, costing us only our pride in branded naming rights. To Paris Carbone for a generous retrospective; your footnote on the VLDB slide deck made Tyler's year, and your shout-out to our what/where/when/how schtick was pure gold. Thanks to our original co-authors: Robert Bradshaw, Craig Chambers, Slava Chernyak, Sam McVeety, Frances Perry, Eric Schmidt, and Sam Whittle. Thanks to the MillWheel, FlumeJava, and Cloud Dataflow teams, and to the Apache Beam community who carried this work into the world. And a sincere thank you to the VLDB community for over a decade of engagement and for the honor.

REFERENCES

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Uğur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. 2005. The Design of the Borealis Stream Processing Engine. In *Proceedings of the Second Biennial Conference on Innovative Data Systems Research (CIDR)*. 277–289.

[2] Atul Adya, Phil Bogle, and Colin Meek. 2025. Understanding the Limitations of Pubsub Systems. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems (HotOS)*. 165–171. https://doi.org/10.1145/3713082.3730397

[3] Atul Adya, Barbara Liskov, and Patrick E. O'Neil. 2000. Generalized Isolation Level Definitions. In *Proceedings of the 16th International Conference on Data Engineering (ICDE)*. 67–78. https://doi.org/10.1109/ICDE.2000.839388

[4] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. 2012. DBToaster: Higher-Order Delta Processing for Dynamic, Frequently Fresh Views. *Proc. VLDB Endow.* 5, 10 (2012), 968–979. https://doi.org/10.14778/2336664.2336670

[5] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *Proc. VLDB Endow.* 6, 11 (2013), 1033–1044. https://doi.org/10.14778/2536222.2536229

[6] Tyler Akidau, Paul Barbier, Istvan Cseri, Fabian Hueske, Tyler Jones, Sasha Lionheart, Daniel Mills, Dzmitry Pauliukevich, Lukas Probst, Niklas Semmler, Dan Sotolongo, and Boyuan Zhang. 2023. What's the Difference? Incremental Processing with Change Queries in Snowflake. *Proc. ACM Manag. Data* 1, 2, Article 196 (2023), 196:1–196:27 pages. https://doi.org/10.1145/3589776

[7] Tyler Akidau, Edmon Begoli, Slava Chernyak, Fabian Hueske, Kathryn Knight, Kenneth Knowles, Daniel Mills, and Dan Sotolongo. 2021. Watermarks in Stream Processing Systems: Semantics and Comparative Analysis of Apache Flink and Google Cloud Dataflow. *Proc. VLDB Endow.* 14, 12 (2021), 3135–3147. https://doi.org/10.14778/3476311.3476389

[8] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proc. VLDB Endow.* 8, 12 (2015), 1792–1803. https://doi.org/10.14778/2824032.2824076

[9] Tyler Akidau, Slava Chernyak, and Reuven Lax. 2018. *Streaming Systems: The What, Where, When, and How of Large-Scale Data Processing*. O'Reilly Media.

[10] Amazon Web Services. 2012. Amazon Simple Workflow Service. https://aws.amazon.com/swf/.

[11] Apache Beam. 2026. Schemas — Beam Programming Guide. https://beam.apache.org/documentation/programming-guide/#what-is-a-schema.

[12] Apache Flink. 2024. Materialized Tables (FLIP-435). https://nightlies.apache.org/flink/flink-docs-release-1.20/docs/dev/table/materialized-table/overview/.

[13] Apache Flink. 2026. Stateful Functions. https://nightlies.apache.org/flink/flink-statefun-docs-master/.

[14] Apache Flink. 2026. Table API. https://nightlies.apache.org/flink/flink-docs-stable/docs/dev/table/tableapi/.

[15] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal* 15, 2 (2006), 121–142. https://doi.org/10.1007/s00778-004-0147-z

[16] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*. 601–613. https://doi.org/10.1145/3183713.3190664

[17] Roger S. Barga, Jonathan Goldstein, Mohamed H. Ali, and Mingsheng Hong. 2007. Consistent Streaming Through Time: A Vision for Event Stream Processing. In *Proceedings of the Third Biennial Conference on Innovative Data Systems Research (CIDR)*.

[18] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth Knowles. 2019. One SQL to Rule Them All: An Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*. 1757–1772. https://doi.org/10.1145/3299869.3314040

[19] Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. 2023. DBSP: Automatic Incremental View Maintenance for Rich Query Languages. *Proc. VLDB Endow.* 16, 7 (2023), 1601–1614. https://doi.org/10.14778/3587136.3587137

[20] Cambra. 2026. Composition Shouldn't Be This Hard. https://cambra.dev/blog/announcement.

[21] Cambra. 2026. The System as a Program. https://cambra.dev/blog/the-system-as-a-program.

[22] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin* 38, 4 (2015), 28–38.

[23] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C. Platt, James F. Terwilliger, and John Wernsing. 2014. Trill: A High-Performance Incremental Query Processor for Diverse Analytics. *Proc. VLDB Endow.* 8, 4 (2014), 401–412. https://doi.org/10.14778/2735496.2735503

[24] Badrish Chandramouli, Jonathan Goldstein, and David Maier. 2009. On-the-Fly Progress Detection in Iterative Stream Queries. *Proc. VLDB Endow.* 2, 1 (2009), 241–252. https://doi.org/10.14778/1687627.1687655

[25] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. 2003. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR)*. https://cidrdb.org

[26] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. 2016. Realtime Data Processing at Facebook. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*. 1087–1098. https://doi.org/10.1145/2882903.2904441

[27] Databricks Inc. 2022. Simplifying Change Data Capture with Databricks Delta Live Tables. https://www.databricks.com/blog/2022/04/25/simplifying-change-data-capture-with-databricks-delta-live-tables.html.

[28] Databricks Inc. 2026. Triggered vs. Continuous Pipeline Mode. https://docs.databricks.com/aws/en/ldp/pipeline-mode. Documentation for Delta Live Tables, since renamed Lakeflow Declarative Pipelines.

[29] DBOS Inc. 2026. DBOS. https://www.dbos.dev.

[30] Tom DeMarco. 1979. *Structured Analysis and System Specification.* Yourdon Press.

[31] Rafael J. Fernández-Moctezuma, Kristin Tufte, and Jin Li. 2009. Inter-Operator Feedback in Data Stream Management Systems via Punctuation. In *Proceedings of the Fourth Biennial Conference on Innovative Data Systems Research (CIDR).*

[32] Google Cloud. 2025. Time Series Functions. https://docs.cloud.google.com/bigquery/docs/reference/standard-sql/time-series-functions#changes.

[33] Google Cloud. 2026. BigQuery: Manage Materialized View Staleness (`max_staleness`). https://cloud.google.com/bigquery/docs/materialized-views-create.

[34] Google Cloud. 2026. Introduction to Continuous Queries. https://docs.cloud.google.com/bigquery/docs/continuous-queries-introduction.

[35] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. 1993. Maintaining Views Incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*. 157–166. https://doi.org/10.1145/170035.170066

[36] Julian Hyde. 2016. Streams, Joins, and Temporal Tables. https://s.apache.org/streams-joins-and-temporal-tables.

[37] Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 194–206. https://doi.org/10.1145/512927.512945

[38] Martin Kleppmann. 2016. *Making Sense of Stream Processing.* O'Reilly Media.

[39] Jay Kreps. 2013. The Log: What Every Software Engineer Should Know about Real-Time Data's Unifying Abstraction. https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying.

[40] Jay Kreps. 2014. Questioning the Lambda Architecture. https://www.oreilly.com/radar/questioning-the-lambda-architecture/.

[41] Krishna Kulkarni and Jan-Eike Michels. 2012. Temporal Features in SQL:2011. *ACM SIGMOD Record* 41, 3 (2012), 34–43. https://doi.org/10.1145/2380776.2380786

[42] Wilburt Labio and Hector Garcia-Molina. 1996. Efficient Snapshot Differential Algorithms for Data Warehousing. In *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB)*. 63–74.

[43] Reuven Lax. 2017. After Lambda: Exactly-Once Processing in Google Cloud Dataflow. https://cloud.google.com/blog/products/data-analytics/after-lambda-exactly-once-processing-in-google-cloud-dataflow-part-1.

[44] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. 311–322. https://doi.org/10.1145/1066157.1066193

[45] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. 2008. Out-of-Order Processing: A New Architecture for High-Performance Stream Systems. *Proc. VLDB Endow.* 1, 1 (2008), 274–288. https://doi.org/10.14778/1453856.1453890

[46] Nathan Marz. 2011. How to Beat the CAP Theorem. http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html.

[47] Materialize Inc. 2026. Isolation Levels. https://materialize.com/docs/reference/isolation-level/.

[48] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. 2020. Shared Arrangements: Practical Inter-Query Sharing for Streaming Dataflows. *Proc. VLDB Endow.* 13, 10 (2020), 1793–1806. https://doi.org/10.14778/3401960.3401974

[49] Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *Proceedings of the Sixth Biennial Conference on Innovative Data Systems Research (CIDR).*

[50] Erik Meijer. 2012. Your Mouse Is a Database. *Commun. ACM* 55, 5 (May 2012), 66–73. https://doi.org/10.1145/2160718.2160735

[51] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems* 17, 1 (1992), 94–162. https://doi.org/10.1145/128765.128770

[52] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. 439–455. https://doi.org/10.1145/2517349.2522738

[53] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. 2017. Samza: Stateful Scalable Stream Processing at LinkedIn. *Proc. VLDB Endow.* 10, 12 (2017), 1634–1645. https://doi.org/10.14778/3137765.3137770

[54] Restate. 2026. Restate. https://restate.dev.

[55] Matthias J. Sax, Guozhang Wang, Matthias Weidlich, and Johann-Christoph Freytag. 2018. Streams and Tables: Two Sides of the Same Coin. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics (BIRTE)*. 1–10. https://doi.org/10.1145/3242153.3242155

[56] Richard T. Snodgrass and Ilsoo Ahn. 1985. A Taxonomy of Time in Databases. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*. 236–246. https://doi.org/10.1145/318898.318921

[57] Snowflake Inc. 2026. Dynamic Tables. https://docs.snowflake.com/en/user-guide/dynamic-tables-about.

[58] Snowflake Inc. 2026. Frozen Regions and Backfill. https://docs.snowflake.com/en/user-guide/dynamic-tables/frozen-regions.

[59] Daniel Sotolongo, Daniel Mills, Tyler Akidau, Anirudh Santhiar, Attila-Péter Tóth, Ilaria Battiston, Ankur Sharma, Botong Huang, Boyuan Zhang, Dzmitry Pauliukevich, Enrico Sartorello, Igor Belianski, Ivan Kalev, Lawrence Benson, Leon Papke, Ling Geng, Matt Uhlar, Nikhil Shah, Niklas Semmler, Olivia Zhou, Saras Nowak, Sasha Lionheart, Till Merker, Vlad Lifliand, Wendy Grus, Yi Huang, and Yiwen Zhu. 2025. Streaming Democratized: Ease Across the Latency Spectrum with Delayed View Semantics and Snowflake Dynamic Tables. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion)*. https://doi.org/10.1145/3722212.3724455

[60] Michael Stonebraker and Uğur Çetintemel. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*. IEEE, 2–11. https://doi.org/10.1109/ICDE.2005.1

[61] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. 2015. General Incremental Sliding-Window Aggregation. *Proc. VLDB Endow.* 8, 7 (2015), 702–713. https://doi.org/10.14778/2752939.2752940

[62] Temporal Technologies. 2019. Temporal. https://temporal.io.

[63] Pınar Tözün, Viktor Leis, Anja Gruenheid, Paris Carbone, and Eleni Tzirita Zacharatou. 2025. Reminiscences on Influential Papers. *ACM SIGMOD Record* 54, 3 (2025), 22–27. https://doi.org/10.1145/3774303.3774307

[64] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. 2003. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Transactions on Knowledge and Data Engineering* 15, 3 (2003), 555–568. https://doi.org/10.1109/TKDE.2003.1198390

[65] Peter A. Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2002. *NEXMark—A Benchmark for Queries over Data Streams (Draft).* Technical Report. OGI School of Science & Engineering at OHSU.

[66] Uber Technologies. 2017. Cadence Workflow. https://cadenceworkflow.io.

[67] Juliane Verwiebe, Philipp M. Grulich, Jonas Traub, and Volker Markl. 2023. Survey of Window Types for Aggregation in Stream Processing Systems. *The VLDB Journal* 32, 5 (2023), 985–1011. https://doi.org/10.1007/s00778-022-00778-6

[68] Ritwik Yadav, Supun Abeysinghe, Min Yang, Jeffrey Helt, Manuel Ung, Yuhong Chen, Melody Hu, William Wei, Yiming Yang, Tom van Bussel, Sourav Chatterji, Indrajit Roy, Paul Lappas, Yannis Papakonstantinou, Tahir Fayyaz, Bilal Aslam, Ross Bunker, Michael Armbrust, and Shrikanth Shankar. 2026. Enzyme: Incremental View Maintenance for Data Engineering. In *Companion of the 2026 International Conference on Management of Data (SIGMOD-Companion)*. https://doi.org/10.1145/3788853.3803098

[69] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. 423–438. https://doi.org/10.1145/2517349.2522737