Building Engines and Platforms for the Big Data Age

Badrish Chandramouli(✉)

Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA
badrishc@microsoft.com

Abstract. Big data analytics involves the collection of real-time operational data into large clusters, followed by the execution of analytics queries to derive insights from the data. The results of these insights are periodically deployed into the real-time pipeline, in order to perform business actions or raise alerts. We are currently witnessing a move towards *fast data analytics*, where some of the offline activities may be performed in memory, directly over the real-time input streams, in order to reduce the time taken to derive and exploit insights from the data. Further, there is an increasing emphasis on enabling data scientists to derive quick approximate insights from large volumes of offline data interactively and at low cost, i.e., without having to process the entire dataset each time. Such hybrid and interconnected workflows across offline and real-time data, stored and processed across multiple machines, and with varying latency needs and complex application logic, requires a rethinking of both data and query processing models and software artifacts that realize such models. This paper surveys the challenges and requirements created by such workflows, and summarizes our research efforts on addressing these problems.

Keywords: Big data · Streaming · Fast data · Analytics · Performance · Latency · Programming languages · Query processing · Iterative · Incremental

1 Introduction

Traditional big data analytics involves the monitoring and archiving of real-time operational data into large cluster storage such as HDFS, Azure Blobs, and SCOPE/Cosmos [16]. This is followed by the execution of offline analytics queries to derive insights from the massive data. The results of these insights are then deployed into the real-time pipeline in order to control and perform business actions or raise alerts, thereby deriving value from the collected data.

Cloud applications that follow this data-centric business model include Web advertising, recommender systems, financial risk analysis, online gaming, and call-center

B. Chandramouli—This paper is based on joint work with Jonathan Goldstein, Mike Barnett, Rob DeLine, Danyel Fisher, John C. Platt, James F. Terwilliger, John Wernsing, Justin Levandoski, Suman Nath, Ivo Santos, Songyun Duan, Wenchao Zhou, Abdul Quamar, and several other collaborators.

analytics. We describe two such applications next, recommender systems and behavior targeted advertising.

Recommender Systems [2, 17]. Platforms such as NetFlix, Reddit, Google News, and Microsoft Xbox need to recommend movies, news items, blog posts etc. to customers. They monitor and archive user ratings of items (e.g., movie ratings or news "likes"). They analyze the collected data by building similarity models (e.g., between users and items) using platforms such as Hadoop. The similarity models feed periodically into a real-time scoring platform (e.g., Storm) that provides recommendations to users by applying the model to that user's current ratings and preferences.

Behavior Targeted Advertising [20, 24]. Advertising platforms such as Google Doubleclick and Bing Ads show targeted ads to users based on their historical behavior. They collect and archive user behavior in the form of ad (advertisement) impression and ad click logs, search history, and URLs visited. They analyze the data to eliminate automated bots, remove spurious clicks, reduce dimensionality, and build machine learning models for users. Finally, during live operations, they track recent per-user behavior in real time and, given an opportunity to show an ad, score the user in real-time and display the most relevant ad.

1.1 New Application Requirements

With increasing quantities of acquired data and the value of timely analytics over the data, Cloud applications such as those outlined above are seeing new requirements that need to be supported by modern data processing engines and platforms. We outline some of these requirements below.

1.1.1 Fast Analytics Over Varied Data Sources

We have recently been witnessing a shift towards *fast data analytics*, where all the activities including data acquisition, analytics or model building, and scoring are performed directly over the real-time streams. This approach dramatically reduces the responsiveness and time to insight of the business, thereby deriving better value from new data as it is collected. For instance, a real-time recommender system can suggest news articles or blog posts within seconds, in time for them to be relevant. An advertising platform can use rapidly changing trends (e.g., flash sales or unexpected events) to target users. The data sources may also be diverse ranging from Cloud-and browser-generated data to device data (e.g., GPS and accelerometer readings) from smartphones, cars, and game consoles. Thus, there is a need to provide users with a powerful language for expressing their complex activities, and a low-overhead runtime that can execute their activities at low latency and directly over real-time streams, potentially distributing the computation across the Cloud and devices.

1.1.2 Unification of Real-Time and Offline

Apart from pushing more processing work to real-time, there is also a strong need to unify the expression and execution of analytics across real-time and offline data sources. We provide several examples of the need for such a unification below:

(1) We may wish to correlate real-time data with events that occurred seven days back in our historical log, in order to detect and report anomalies, defined as significant deviations from expected behavior.
(2) We may wish to "back-test" real-time queries over historical logs. For example, we may have a real-time query deployed in production, and would like to test or execute over offline data, perhaps with different parameter settings, without having to rewrite the query logic. Efficiency of execution is particularly critical during offline execution, due to the large quantities of data being analyzed.
(3) We may wish to take the results of our offline analysis and operationalize them (i.e., deploy them in real-time) without having to change the query logic.

All of these activities need an expressive query model and platforms that can execute that query model efficiently over real-time and/or offline datasets.

1.1.3 Interactive Exploration

The data acquired in real time is usually stored in large clusters for offline analysis by data scientists. Analytics over the large volumes of collected data can be very expensive. The pay-as-you-go paradigm of the Cloud causes computation costs to increase linearly with query execution time, making it possible for a data scientist to easily spend large amounts of money analyzing data. The problem is exacerbated by the interactive and exploratory nature of analytics, where queries are iteratively discovered and refined, including the submission of many off-target and erroneous queries (e.g., bad parameters). In traditional systems, queries must execute to completion before such problems are diagnosed, often after hours of expensive compute time are exhausted.

We define *progressive analytics* as the generation of early results to analytical queries based on partial data, and the progressive refinement of these results as more data is received. Progressive analytics allows users to get early results using significantly fewer resources, and potentially end (and possibly refine) computations early once sufficient accuracy or query incorrectness is observed. We need query models and runtimes that can support workflows that include such progressive analytics over offline datasets, in addition to being able to process time-oriented queries on real-time and/or offline data. The progressive queries themselves should be easy to author by data scientists in an interactive and visual manner, and should be easy to operationalize into a real-time pipe.

1.2 Today's Solutions

Partly as a result of the diverse and inter-connected nature of analytics as outlined above, the big data analytics landscape of today is quite rich in terms of available systems for performing data processing. Database systems such as Vertica [44] and Microsoft SQL Server [39] are used for relational analytics. Map-reduce systems such as Hadoop [6] and its variants [34] are used for partitioned queries over offline data, with front-ends such as Hive to support relational queries. Spark [46] (and Spark SQL) offer high-performance data transformations over offline data that may be cached in main memory. Separate systems such as GraphLab [37] support iterative queries for graph analytics. S-STORE [15] provides a unified engine to handle transaction processing and streaming computation. Systems such as BlinkDB [13] and CONTROL [29, 30] enable approximate early results for analytics queries. Stream processing engines such as STREAM [8], Borealis [1], NiagaraST [38], Nile [28], DataCell [35], and Microsoft StreamInsight [5] are used for running real-time queries. Distributed streaming platforms such as Storm [7], Spark Streaming [47], MillWheel [4], and Naiad [40] are used for distributed stream processing. The Berkeley Data Analytics Stack [11] provides several tools and systems to perform analytical, graph, streaming, and transactional computations.

The data platforms landscape map [26] illustrates the enormous number of platforms and systems that now exist in the big data ecosystem, with disparate tools, data formats, and techniques [36]. Combining these disparate tools with application-specific glue logic in order to execute end-to-end workflows is a tedious and error-prone process, with potentially poor performance and the need for translation at each step. Further, the lack of a unified data model and query semantics precludes reusing the same logic across all the tools, handling cross-platform query processing (e.g., across devices and the Cloud), developing queries on historical data and then deploying them directly to live streams, or back-testing live queries (with possibly different parameters) on historical datasets.

1.3 Towards a Unified Analytics Solution

Over the last several years, we have been working on developing and refining models, methods, and system architectures to help alleviate the range of challenges outlined above. Starting from query and data models all the way to concrete system designs and architectures, we have been rethinking the way we build engines and platforms so that Cloud applications can support the new requirements, without having to suffer the impedance mismatch introduced by simply putting together complex and diverse technologies. Our research work spans the areas of semantics, engines, and platforms:

We have worked on unifying the use of the tempo-relational model and its refinements for big data [9, 20, 21, 23, 33, 38], significantly improving upon semantic clarity, expressiveness, and algorithms in an incremental setting;

We have built temporal streaming engines that realize such models, such as:

- StreamInsight [5], which shipped as part of Microsoft SQL Server;
- Trill [19], a. NET-based engine that provides best-of-breed or better performance across the entire latency spectrum; and
- JStreams [42], a Javascript-based temporal engine for Web browsers and devices.

Our engines are designed with a goal of running seamlessly on different scale-out platforms with varying latency goals. We have demonstrated this layered approach via systems such as:

- TiMR [20], which allows us to embed an unmodified streaming engine within an unmodified map-reduce system for offline temporal analytics;

- Race [18], which enables real-time streaming queries to efficiently execute across *edge devices* (e.g., smartphones) and the Cloud; and
- Now! [23], which enables an unmodified streaming engine to run progressive relational queries in a pipelined map-reduce setting.

The goal of this paper is to summarize our key insights and learnings in the process of providing new and unified data processing solutions for the complex big data analytics landscape. We start in Sect. 2 by describing a unified query model that can support all of the complex workflow requirements outlined earlier. In Sect. 3, we focus on the runtime, and discuss several key system requirements, architectural choices, and solutions that can enable big data applications to execute their workflows seamlessly and efficiently while leveraging and reusing the breadth of platforms available today. We use a case study in Sect. 4 to demonstrate the use of these tools to solve data processing problems in the context of Web search and advertising, and conclude the paper in Sect. 5.

2 Choosing a Query Model for the Big Data Age

We need a query model that can effectively support the range of analytics described above. The standard relational model used in database systems is expressive, but does not handle low-latency real-time queries over changing data. Support for incremental iterative processing is necessary to handle queries over changing graphs. Further, the data in our applications is temporal in nature, i.e., time is a first class citizen. For example, Web advertising queries operate on reports of ad clicks, ad impression, and Web searches by users; each of these activities is associated with a timestamp of occurrence. Given the temporal nature of data, many application activities consist of queries that have a fundamental temporal component to them. For example, the first step of Web advertising usually consists of eliminating the influence of black-listed *bot users*, who are defined as spurious users (usually automated clickers) who, at a given instant of time, have clicked on more than a specified threshold of ads within a short time period. The tempo-relational data and query model, with appropriate extensions for streaming computation, turns out to be a very good fit for handling the entire range of analytics described above, and can serve as the backbone for big data analytics. We describe this model next.

2.1 The Tempo-Relational Data and Query Model

A *streaming engine* enables applications to execute long-running *continuous queries* (*CQs*) over data streams in real-time. Streaming engines are used for efficient real-time processing in applications such as fraud detection, monitoring RFID readings from sensors, and algorithmic stock trading. While streaming engines target real-time data, they are usually based on some variant of the tempo-relational data and query model from early work on temporal databases.

2.1.1 Data Model

Logically, we can view a stream as a *temporal database* (*TDB*) [33] that is presented incrementally, as in CEDR [9], Nile [28], NiagaraST [38], etc. The TDB is a multi-set of logical *events*, each of which consists of a relational tuple p (called the payload) and an associated validity interval denoted by a validity start time V_s and a validity end time V_e which define a half-open interval $[V_s, V_e)$. One can think of V_s as representing the event's timestamp, while the validity interval is the period of time (or *data window*) over which the event is active and contributes to output. In the tempo-relational model, the dataset gets divided into *snapshots*, a sequence of data versions across time, that are formed by the union of unique interval endpoints of all events in the input TDB. In Fig. 1 (left), we show three interval events that divide the timeline into five snapshots.

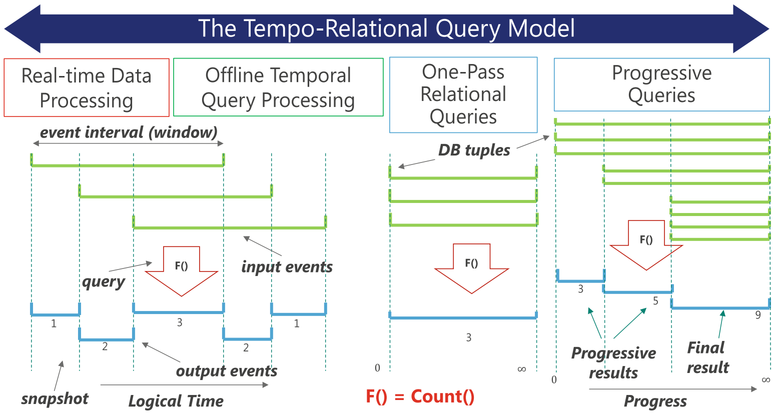

Fig. 1. Expressive power of the tempo-relational query model.

Physically, different streaming engines make different choices with respect to stream contents. For example, STREAM represents events as I-streams and D-streams. In Trill, events may either arrive directly as an *interval*, or get broken up into a separate insert into (called *start-edge*) and delete from (called *end-edge*) the TDB. A special case of interval events is a *point* event $[V_s, V_s + 1)$, which has a lifetime of one *chronon* (the smallest possible unit of time), and represents an instantaneous occurrence of an event. Punctuations [38] are used in systems such as NiagaraST, StreamInsight, and Trill to denote the progress of time in the absence of data. Further, systems such as Trill batch events for higher performance, and therefore use punctuations for the secondary purpose of forcing the flushing of partial batches from the engine.

2.1.2 Query Model

Users can express a query Q over the timestamped data. The query is specified using a language such as StreamSQL [32] or temporal LINQ [43], and is compiled into a graph of streaming operators. Logically, Q maps one TDB instance to another: the query Q is logically executed over every snapshot. A result is present in the output TDB at timestamp T if it is the result of Q applied to data that is "alive at" (whose validity intervals are stabbed by) T. Figure 1 (left) shows a Count query that outputs, for each snapshot, a count of events in that snapshot. Physically, the output TDB is computed and emitted incrementally as input events are presented to operators in the engine, usually in timestamp order.

2.1.3 Temporal Operators

All standard relational operations such as *Filter*, *Project*, *Union*, *Join*, *Aggregation*, and set difference with *AntiSemiJoin* – also called *WhereNotExists* – have equivalent temporal counterparts. The *SelectMany* operator introduced in LINQ maps each row to zero or more transformed output rows. A *Multicast* operator is used to copy an input stream into more than one output stream. The class of stream operators that incrementally process stream elements and produce per-snapshot output are called snapshot operators (e.g., in-built and user-defined aggregates). Most engines also have the ability to process grouped computations with the *GroupApply* operation [20], which logically executes a sub-query on every sub-stream consisting of events with the same grouping key value. Streaming engines usually also support sequential pattern detection operators to detect complex regular-expression-style sequences of event occurrences [22, 45]. These are useful, for example, to detect complex chart patterns (e.g., the *candlestick* pattern) in algorithmic stock trading.

Further, streaming engines include incremental operators that can manipulate time in safe ways. The *Window* operator sets the lifetime of each event to a specified value, and can be used to window the data, for example, to output a sliding-window aggregate over the last 5 min. The start time of events may also be modified to, for instance, create hopping windows. *Clip* is a special data-dependent windowing operator that takes two input streams. It windows the data on the left stream based on matching future data in the right stream. For example, we can clip a start-session stream with an end-session stream to generate a sequence of session events, each with a start and end time that correspond to session start and end timestamps respectively. In Sect. 5, we will use a case study of Web advertising to show how some of these operators can be used to answer complex questions over temporal data.

2.2 Temporal, Relational, and Iterative Query Support

Since the tempo-relational model is based on application time, query results are a function of the data and query alone, and are not dependent on the wall-clock time of actual execution. This allows us to seamlessly execute real-time queries on historical logs and vice versa. Further, we can meaningfully execute queries that correlate real-time data with historical data using application time.

Further, it is easy to see that this model is a superset of the relational model, as shown in Fig. 1 (center). By setting all events to have the same time interval (say, $[0, \infty)$), we create a single snapshot that represents the execution of a standard relational query. Further, iterative queries (both relational and temporal) can be supported by adding looping support to the query plan, taking additional care to detect and propagate the progress of time in an incremental setting [21].

2.3 Progressive Query Support

We have shown in prior work [23] that one can use the tempo-relational model to also support progressive relational queries. The key idea, shown in Fig. 1 (right), is to re-interpret application time to instead mean query computation progress. Based on a user-specified (or system generated) data sampling strategy, we simply timestamp the data as $[0, \infty), [1, \infty), \ldots$ to denote (usually increasing) subsets of data in a series of snapshots. Executing a query using an unmodified temporal streaming engine, over data annotated in this manner, results in the generation of early results that are refined as more data is processed. For example, Fig. 1 (right) shows the results of a simple relational Count query as early results 3 and 5, that are finally refined to the exact (final) value of 9. This technique provides determinism and repeatability for early results – two runs of a query provide the same sequence of early answers. This greatly enhances our understanding and debuggability of early results.

Sampling strategies are simply encoded as timestamp assignments. For example, with a star-schema, we may set all tuples in the (small) dimension table to have an interval of $[0, \infty)$, while progressively sampling from the fact table as $[1, \infty)$, $[2, \infty), \ldots$, which effectively causes a temporal Join operator to fully "preload" the dimension table before progressively sampling the fact table for meaningful early results.

3 System Designs for the Big Data Age

Based on our query and data model, we now describe key design and architectural choices that need to be made when creating a unified analytics stack that can handle a variety of analytics scenarios. Our proposed approach consists of first building a general-purpose and powerful streaming query engine as a reusable component. Such an engine is then reused in different settings, and embedded within a variety of platforms to handle different user scenarios.

3.1 Design Considerations for a Streaming Engine as a Component

3.1.1 Performance and Overhead

A single streaming engine used for a variety of analytics scenarios outlined in this paper requires very high and best-of-breed performance across the latency spectrum from offline to real-time. High performance also translates to low overhead, which can be critical in monitoring applications, and when stream processing is offloaded to devices. As an example, batching is becoming a standard technique used by engines such as Naiad, Storm (with the Trident API), Spark Streaming, and Trill to gain high performance. In the same vein, systems such as Trill expose a dynamic latency-throughput tradeoff to users in order to provide high performance across the latency spectrum, and adopt several database-style optimizations such as columnar processing to provide even higher performance gains.

3.1.2 Server Versus Library Execution

We find that the traditional server model of databases – where the database "owns" the machine (or set of machines), manages thread scheduling, and uses native memory management with a rigid type system (e.g., SQL types) – is a poor fit for streaming in the big data age, due to several reasons. The Cloud application is usually in control of the end-to-end application, and invokes the engine as part of its data processing. Complex data-types such as machine learning models often need to be supported and streamed through the engine. Further, user-defined extensions and libraries are very common, and are usually written using high-level languages. Such extensions and libraries need to seamlessly integrate at with query processing without loss of performance or a need for fine-grained data transformations. Further, threads are often already managed and owned by scale-out fabrics, and thus do not inter-operate well with engines that try to take on such a role. As a result, many streaming engines offer deep high-level language support (e.g., Naiad, Storm, Spark Streaming, and Trill) or can optionally execute as a library that does not own threads (e.g., Rx [41] and Trill). Finally, the engine also needs to expose operator state checkpointing capabilities that can be leveraged by the surrounding platforms to achieve resiliency to failure.

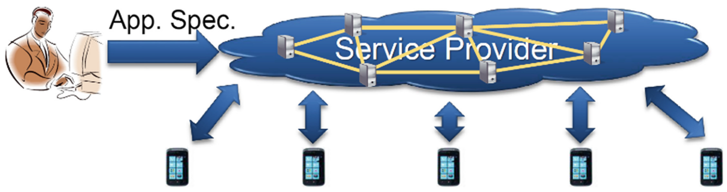

Fig. 2. Architecture of Cloud-Edge applications.

3.2 Platforms for Big Data Analytics

As discussed earlier, a temporal streaming engine can be embedded within a variety of distribution platforms to handle various scenarios. We overview some of the ways we have enabled specific requirements in the past using this layering approach.

3.2.1 Cloud-Edge Applications Across Real-Time and Offline Data

Figure 2 shows the architecture of a typical Cloud-Edge application. Data generated in the Cloud and by edge devices such as smartphones (such as GPS readings), as well as offline reference data (such as social network graphs) is made available for querying by the Cloud application. For example, a social Foursquare application may wish to notify a user whenever any of her friends checks in or visits a nearby location. Such applications can be expressed as streaming queries, and the corresponding query graph can be partitions to execute on edge devices and in the Cloud. In order to accomplish this, we can leverage a hybrid deployment of streaming engines based on a common tempo-relational model, but targeted towards different platforms. For instance, we could combine Trill running in the Cloud with JStreams – a temporal streaming engine written in Javascript – running on smartphones. Devices may have constraints on capabilities (e.g., CPU, memory, battery usage) as well as data upload (e.g., due to bandwidth costs or privacy settings). Deciding which operators execute at which location is an optimization problem that is hard in general, but needs to be addressed in this hybrid setting; see [18] for some solutions we have proposed in this space. The real-time processing component that executes in the Cloud may itself need to be scaled out, as we discuss next.

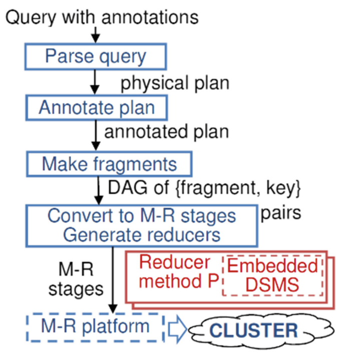

Fig. 3. TiMR design.

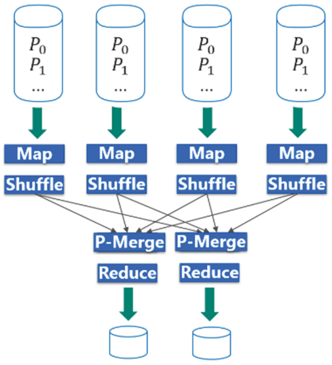

Fig. 4. Now! architecture.

3.2.2 Real-Time Scaled-Out Processing in the Cloud

Unlike offline data processing, very high event rates are usually not a requirement for real-time analytics. For example, most internet activity such as searches and tweets consist of fewer than 100 K events per second [31]. However, scale out is still necessary for the scalable ingress of data, and because many real-time processing queries is memory constrained. For example, we may need to track the activity of millions of users in a real-time advertising platform. Orleans [12] is a programming model and fabric that enables low-latency (in milliseconds) distributed streaming computations with units of work called grains. Orleans owns threads and manages the distributed execution of user code, but offers no declarative language or query processing capabilities. Streaming libraries such as Rx or Trill are often embedded within Orleans grains in order to execute real-time queries, thus allowing applications to scale their real-time computation on to multiple machines; [14] describes such an application. Another example of such a fabric that can embed a streaming engine is REEF [25].

3.2.3 Temporal Analytics on Offline Datasets

The acquired real-time data is usually stored in large storage clusters such as HDFS and Cosmos. We may wish to take our real-time queries and execute them on the offline data. Map-reduce (M-R) is a common paradigm for executing queries on large offline datasets. With the streaming engine as a library, one can easily embed the engine within reducers in map-reduce, in order to execute the same temporal queries on the offline data. We built a framework called TiMR (pronounced timer), to process temporal queries over large volumes of offline data [20]. Figure 3 depicts the design of TiMR.

Briefly, TiMR combines an unmodified data stream management system (DSMS) with an unmodified map-reduce distributed computing platform. Users perform analytics using a temporal language (e.g., temporal LINQ or StreamSQL). The query DAG (directed acyclic graph) is converted into map-reduce jobs that run efficiently on large-scale offline temporal data in a map-reduce cluster. Further, the queries are naturally ready to operationalize over real-time data. TiMR leverages the tempo-relational model underlying the DSMS for repeatable behavior across runs.

Time:long	UserId:string	AdId:string

(a) Impression Logs

Time:long	UserId:string	AdId:string

(b) Click Logs

Time:long	UserId:string	Keyword:string

(c) Search and Page View Logs

Time:long	StreamId:int	UserId:string	KwAdId:string

(d) Unified schema for BT data.

Fig. 5. Data schema for behavior targeted advertising.

3.2.4 Enabling Progressive Analytics for Data Scientists

Recall that the tempo-relational model can support progressive queries by re-interpreting time to mean computation progress. We built a distributed platform called Now! (see Fig. 4), which improves upon a standard map-reduce framework to include pipelining and support for time (or more accurately, progress) as a first-class citizen within the framework. Consider a large progress-annotated dataset, that is split into a sequence of partitioned *progress batches* P_i, each of which represents a chunk of input data with the same validity interval of $[i, \infty)$. Now! understands that data consists of a sequence of progress batches; therefore, after the map phase, its shuffle operation creates a sequence of progress batches per destination reducer. On the reduce side, the incoming progress batches from multiple mappers are merged in timestamp order (using the P-Merge operator shown in Fig. 4), before being fed to a *progressive reducer* in a pipelined timestamp-ordered fashion. With this architecture, we were able to embed an unmodified streaming engine into the reducer as in TiMR, and execute large-scale queries progressively. More details on Now! can be found in our research paper [23].

From a usability perspective, we built a Web-based front end for issuing progressive queries, called Tempe (formerly Stat!) [10, 27], which allows data scientists to collaboratively author queries with progressive processing and visualization.

4 Case Study: Web Advertising

Behavior targeted advertising operates on data with the (simplified) data format shown in Fig. 5(a–c). Each ad impression (or click) entry has a timestamp, the id of the user to who was shown (or clicked on) the ad, and the id of the ad itself. The keyword search and page view entries are similar, consisting of a timestamp, the id of the user, and the search term or page URL. This data may be acquired in real-time, and stored in large storage clusters such as HDFS and Scope/Cosmos. For simplicity, we assume a single unified schema for the data, as shown in Fig. 5(d). Here, we use StreamId to disambiguate between the various sources. StreamId values of 0, 1, and 2 refer to ad impression, ad click, and keyword (searches and pageviews) data respectively. Based on StreamId, the column KwAdId refers to either a keyword or an AdId. Our queries are written to target the new schema, and thus operate on a single input data source.

4.1 Temporal Query: Bot Elimination

The goal of bot elimination is to get rid of activity corresponding to users that have "unusual" behavior characteristics. We define a bot as a user who either clicks on more than T_1 ads, or searches for more than T_2 keywords within a time window τ. Before feeding data to training models, it is important to detect and eliminate bots quickly, as we receive user activity information; otherwise, the actual correlation between user behavior and ad click activities can be diluted by the spurious behavior of bots.

The CQ shown in Fig. 6 gets rid of bots. We first create a hopping window (implemented using the Window operator) with hop size h = 15 min and window size w = 6 h, over our input unified point event stream S1. This updates the bot list every 15 min using data from a 6 h window. The GroupApply (with grouping key UserId) applies the following sub-query to each UserId sub-stream. From the click and search input streams for that user, we perform the count operation on each stream, filter out counter events with value less than the appropriate threshold (T_1 or T_2), and use Union to get one stream S2 that retains only bot users' data. We finally perform an AntiSemiJoin (on UserId) of the original stream S1 with S2, to output only non-bot users' data. Note that UserId can serve as the partitioning key for this CQ's execution in a scaled-out setting.

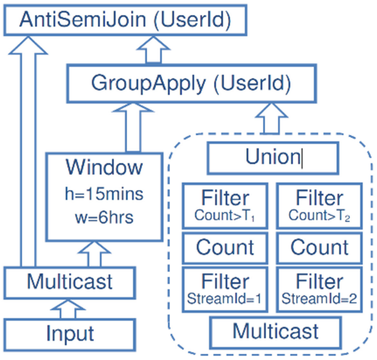

Fig. 6. Bot elimination CQ.

4.1.1 Real-Time Versus Offline Bot Elimination

The bot elimination CQ can be used directly on real-time streams, for example using Trill with Orleans. It can also be executed over offline data using TiMR running over SCOPE/Cosmos. Note that bot elimination is temporal – on offline data, it does not simply create and use a single list of bot users on the entire dataset. Rather, it computes, reports, and filters the input data based on the *varying* set of bot users over time, allowing for users to enter and exit the bot classification over the lifetime of the log.

4.2 Progressive Query: Top-K Correlated Search Terms

Assume we have collected a large quantity of search logs, and a data scientist wishes to analyze the data for effective user targeting. A query they may wish to execute on the dataset is: reports the top-k search terms that are most correlated with an analyst-provided *search parameter* (e.g., "birthday"), according to a goodness score, in the search dataset. The data scientist knows that taking random samples by UserId will yield good approximate answers for queries of this nature, so they timestamp the data accordingly in the cluster, and materialize it sorted by timestamp in the cluster.

The query can be expressed using two GroupApply operations, and is executed as follows using the Now! framework with Trill running within the progressive reducers. The first stage uses the dataset as input and partitions by UserId. Each reducer tokenizes the data into search terms using SelectMany, and uses GroupApply to compute a (partial) histogram that reports, for each search term in that partition, the number of searches with and without the search parameter, and the total number of searches. The next stage groups by search term, aggregates the partial histograms from the first stage reducers, computes a per-search-term goodness. Finally, we use a top-k aggregate to report the search terms that are most closely correlated to the input search parameter. Using an interactive environment such as Tempe, the data scientist can receive immediate results for such a query.

4.3 Cloud-Edge Query: Location-Aware Advertising

Assume we have built a scoring (machine leaning) model in the Cloud that, given the historical searches and page views of a user, can score any advertisement (or coupon) for its relevance to that user. Further, assume that advertisements are for physical stores, and we wish to target users with smartphones such that they receive advertisements relevant to their search history when they are in close proximity (say, within 1 mile) of the corresponding store location. This query involves a join of user searches (from their browser or search apps), their changing locations (from GPS readings on their device), slow-changing reference data of store locations available at the Cloud, and the scoring model which itself may change over time. In such a Cloud-Edge topology, we may wish to execute parts of the join on the users' smartphones by downloading the model and advertising information streams to the device, instead of constantly uploading their locations to the Cloud. This approach can also allow users to receive relevant ads in the presence of device constraints or privacy settings where they prevent their local searches and/or location information from being sent to or stored in the Cloud.

5 Conclusions

Big data analytics involves the collection of real-time data, followed by the execution of analytics queries to derive insights from the data. The results of these insights are deployed into the real-time pipeline, in order to perform business actions or raise alerts. We are witnessing an increasing need for fast data analytics in order to reduce the time to derive insights from data. Further, there exists remarkable diversity in the types of analytics that need to be performed, including latencies (real-time to offline) and settings (such as temporal, relational, iterative, and progressive). In this paper, we observe that the tempo-relational model can form a strong foundation that cuts across this diversity of analytics and provides a unified view of data and queries. We then overview several design choices and system architectures that leverage the tempo-relational model to build unified engines and platforms for big data analytics. Finally, we use a case study of Web advertising to illustrate the value of these models and system design choices.

References

1. Abadi, D.J., et al.: The design of the borealis stream processing system. In: CIDR (2005)
2. Adomavicius, G., Tuzhilin, A.: Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions. TKDE **17**(6), 734–749 (2005)
3. Agarwal, S., et al.: BlinkDB: queries with bounded errors and bounded response times on very large data. In: EuroSys (2013)
4. Akidau, T. et al.: MillWheel: fault-tolerant stream processing at internet scale. In: VLDB (2013)
5. Ali, M. et al.: Microsoft CEP server and online behavioral targeting. In: VLDB (2009)
6. Apache hadoop. http://hadoop.apache.org/
7. Apache storm. http://storm.incubator.apache.org/
8. Babcock, B., et al.: Models and issues in data stream systems. In: PODS (2002)
9. Barga, R.S., et al.: Consistent streaming through time: a vision for event stream processing. In: CIDR, pp. 363–374 (2007)
10. Barnett, M., et al.: Stat! - an interactive analytics environment for big data. In: SIGMOD (2013)
11. Berkeley data analytics stack (BDAS). https://amplab.cs.berkeley.edu/software/
12. Bernstein, P., et al.: Orleans: distributed virtual actors for programmability and scalability. MSR Technical report (MSR-TR-2014-41, 24). http://aka.ms/Ykyqft
13. BlinkDB. http://blinkdb.org/
14. Building real-time services for halo. http://research.microsoft.com/apps/video/?id=198324
15. Cetintemel, U., et al.: S-Store: a streaming new SQL system for big velocity applications. In: VLDB (2014)
16. Chaiken, R., et al.: SCOPE: easy and efficient parallel processing of massive data sets. PVLDB **1**(2), 1265–1276 (2008)
17. Chandramouli, B., Levandoski, J.J., Eldawy, A., Mokbel, M.: StreamRec: a real-time recommender system. In: SIGMOD (2011)
18. Chandramouli, B., Nath, S., Zhou, W.: Supporting distributed feed-following apps over edge devices. PVLDB **6**(13), 1570–1581 (2013)
19. Chandramouli, B., Goldstein, J., Barnett, M., DeLine, R., Fisher, D., Platt, J.C., Terwilliger, J.F., Wernsing, J.: Trill: a high-performance incremental query processor for diverse analytics. In: VLDB (2015, to appear)
20. Chandramouli, B., Goldstein, J., Duan, S.: Temporal analytics on big data for web advertising. In: ICDE (2012)
21. Chandramouli, B., Goldstein, J., Maier, D.: On-the-fly progress detection in iterative stream queries. In: VLDB (2009)

22. Chandramouli, B., Goldstein, J., Maier, D.: High-Performance dynamic pattern matching over disordered streams. In: VLDB (2010)
23. Chandramouli, B., Goldstein, J., Quamar, A.: Scalable progressive analytics on big data in the cloud. PVLDB **6**(14), 1726–1737 (2013)
24. Chen, Y., et al.: Large-scale behavioral targeting. In: KDD (2009)
25. Chun, B., et al.: REEF: retainable evaluator execution framework. PVLDB **6**(12), 1370–1373 (2013)
26. Data platforms landscape map. http://blogs.the451group.com/information_management/2014/11/18/updated-data-platforms-landscape-map/
27. Fisher, D., Chandramouli, B., DeLine, R., Goldstein, J., Aron, A., Barnett, M., Platt, J.C., Terwilliger, J.F., Wernsing, J.: Tempe: an interactive data science environment for exploration of temporal and streaming data. MSR Technical report MSR-TR-2014–148 (2014). http://research.microsoft.com/apps/pubs/?id=232385. Accessed Nov 2014
28. Hammad, M., et al.: NILE: a query processing engine for data streams. In: ICDE (2004)
29. Hellerstein, J.M., Avnur, R.: Informix under control: online query processing. J. Data Min. Knowl. Discov. **12**, 281–314 (2000)
30. Hellerstein, J.M., Haas, P.J., Wang, H.J.: Online aggregation. In: SIGMOD (1997)
31. Internet live stats. http://www.internetlivestats.com/
32. Jain, N., et al.: Towards a streaming SQL standard. In: VLDB (2008)
33. Jensen, C., Snodgrass, R.: Temporal specialization. In: ICDE (1992)
34. Li, B., et al.: A platform for scalable one-pass analytics using MapReduce. In: SIGMOD, pp. 985–996 (2011)
35. Liarou, E., et al.: Enhanced stream processing in a DBMS kernel. In: EDBT (2013)
36. Lim, H., et al.: How to fit when no one size fits. In: CIDR (2013)
37. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Distributed GraphLab: a framework for machine learning in the cloud. In: VLDB (2012)
38. Maier, D., Li, J., Tucker, P., Tufte, K., Papadimos, V.: Semantics of data streams and operators. In: Eiter, T., Libkin, L. (eds.) ICDT 2005. LNCS, vol. 3363, pp. 37–52. Springer, Heidelberg (2005)
39. Microsoft SQL server. http://www.microsoft.com/en-us/server-cloud/products/sql-server/
40. Murray, D., et al.: Naiad: a timely dataflow system. In: SOSP (2013)
41. Reactive extensions for .NET. http://aka.ms/rx
42. Santos, I., Tilly, M., Chandramouli, B., Goldstein, J.: DiAl: distributed streaming analytics anywhere anytime. In: VLDB (2013)
43. The LINQ project. http://aka.ms/rjhi00
44. Vertica. http://www.vertica.com/
45. Wu, E., Diao, Y., Rizvi, S.: High-performance complex event processing over streams. In: SIGMOD (2006)
46. Zaharia, M., et al.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: NSDI (2012)
47. Zaharia, M., et al.: Discretized streams: fault-tolerant streaming computation at scale. In: SOSP (2013)