Temporal Analytics on Big Data for Web Advertising

Badrish Chandramouli[†], Jonathan Goldstein[#], Songyun Duan[‡]
[†]Microsoft Research, Redmond [#]Microsoft Corp., Redmond [‡]IBM T. J. Watson Research, Hawthorne
{badrishc, jongold}@microsoft.com, sduan@us.ibm.com

***Abstract*— "Big Data" in map-reduce (M-R) clusters is often fundamentally temporal in nature, as are many analytics tasks over such data. For instance, display advertising uses *Behavioral Targeting* (*BT*) to select ads for users based on prior searches, page views, etc. Previous work on BT has focused on techniques that scale well for offline data using M-R. However, this approach has limitations for BT-style applications that deal with temporal data: (1) many queries are temporal and not easily expressible in M-R, and moreover, the set-oriented nature of M-R front-ends such as SCOPE is not suitable for temporal processing; (2) as commercial systems mature, they may need to also directly analyze and react to real-time data feeds since a high turnaround time can result in missed opportunities, but it is difficult for current solutions to naturally also operate over real-time streams.**

Our contributions are twofold. First, we propose a novel framework called TiMR (pronounced *timer*), that combines a time-oriented data processing system with a M-R framework. Users perform analytics using *temporal queries* — these queries are succinct, scale-out-agnostic, and easy to write. They scale well on large-scale offline data using TiMR, and can work unmodified over real-time streams. We also propose new cost-based query fragmentation and temporal partitioning schemes for improving efficiency with TiMR. Second, we show the feasibility of this approach for BT, with new temporal algorithms that exploit new targeting opportunities. Experiments using real advertising data show that TiMR is efficient and incurs orders-of-magnitude lower development effort. Our BT solution is easy and succinct, and performs up to several times better than current schemes in terms of memory, learning time, and click-through-rate/coverage.

I. Introduction

The *monitor-manage-mine* (*M3*) loop is characteristic of data management in modern commercial applications. We *monitor* and archive incoming data, that is used to *manage* daily business actions. We *mine* the collected "big data" to derive knowledge that feeds back into the monitor or manage phases.

For example, consider the problem of *display advertising*, where ads need to be shown to users as they browse the Web. *Behavioral Targeting* (*BT*) [34] is a recent technology, where the system selects the most relevant ads to display to users based on their prior behavior such as searches and webpages visited. The system monitors users and builds a *behavior profile* for each user, that consists of their historical behavior. For example, the profile may consist of a count for each page visited or keyword searched in a time-frame. The ad click (and non-click) activity of users, along with their corresponding behavior profiles, are collected and used during the mining phase to build models. The models are used during the operational (manage) phase to score users in real time, i.e., predict the *relevance* of each ad for a current user who needs to be delivered an ad. A common measure of relevance for BT is *click-through-rate* (*CTR*) — the fraction of ad impressions that result in a click [34, 7]. Many companies including Yahoo! SmartAds, Microsoft adCenter, and DoubleClick use BT as a core component of their advertising platform.

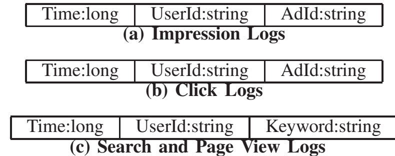

Fig. 1. Schemas for BT data.

Advertisement systems collect and store data related to billions of users and hundreds of thousands of ads. For effective BT, multiple mining steps are performed on the data:

- *Bot Elimination*: We need to detect and eliminate *bots*, which are automated surfers and ad clickers, to eliminate spurious data before further analysis, for more accurate BT.
- *Data Reduction*: The behavior profiles are sparse and of extremely high dimensionality, with millions of possible keywords and URLs. We need to get rid of useless information in a manner that retains and amplifies the most important signals for subsequent operations. Some common data reduction schemes used for BT include (1) mapping keywords to a smaller set of concepts by *feature extraction* [12], and (2) retaining only the most popular attributes by *feature selection* [7].
- *Model Building and Scoring*: We need to build accurate models from the behavior profiles, based on historical information about ad effectiveness. For example, Yan et al. [34] propose grouping similar users using clustering, while Chen et al. [7] propose fitting a Poisson distribution as a model for the number of clicks and impressions.

Challenges and Contributions

In order to scale BT, the historical data is stored in a distributed file system such as HDFS [16], GFS [14], or Cosmos [6]. Systems usually analyze this data using *map-reduce* (*M-R*) [10, 16, 20, 35] on a cluster. M-R allows the same computation to be executed in parallel on different data partitions.

The input data for BT consists of terabytes of logs. We show a relevant subset of columns in these BT logs, in Figure 1. Impression (or click) logs identify the timestamp when a user is shown (or clicks on) an ad, while the search and page view logs indicate when a user performs a search or visits a URL (denoted by Keyword). A crucial observation is that this data is fundamentally *temporal* (i.e., related closely to time). Further, it turns out that many common analytics queries (including BT steps) are also fundamentally temporal, requiring the ability to perform time-ordered sequential processing over the data.

1) Scalability with Easy Specification

As warm-up, consider a very simple BT-style analytics query:

Example 1 (RunningClickCount)**.** *A data analyst wishes to report how the number of clicks (or average CTR) for each ad in a* 6*-hour window, varied over a* 30*-day dataset. This query may be used to determine periodic trends or data correlations.*

This query is temporal in nature. Front-end languages such as Pig [13], SCOPE [6], and DryadLinq [35] are frequently used to make it easier to perform analytics. However, we will see in Section III that queries such as the above involve non-trivial temporal sequence computations that are fundamentally difficult to capture using traditional database-style set-oriented languages (in fact, we will see that the SCOPE query for RunningClickCount is intractable). We could instead write our own customized map-reduce code from scratch, but such customized algorithms are more complex, harder to debug, and not easily reusable (windowed aggregation in Microsoft StreamInsight uses more than 3000 lines of code).

A more complex example is BT, where the analytics is fundamentally temporal, e.g., whenever a user clicks or rejects an ad, we need access to the last several hours of their behavior data *as of that time instant*. This is useful for detecting subtle behavior correlations for BT, as the example below indicates.

Example 2 (Keyword Trends)**.** *A new television series (*iCarly*) targeted towards the teen demographic is aired, resulting in a spike in searches for that show. Interestingly, we found (see Section V) that searches for the show were strongly correlated with clicks on a deodorant ad. Other keywords positively correlated with this ad included "celebrity", "exam", "chat", "music", etc. In contrast, keywords such as "jobless", "credit", and "construction" were negatively correlated with clicks on the same ad.*

The *temporal-analytics-temporal-data* characteristic is not unique to BT, but is true for many other large-scale applications such as network log querying [24], collaborative filtering over community logs [9], call-center analysis, financial risk analysis, and fraud detection. Given the inherent complexity of temporal analytics on big data, we need a mechanism whereby analysts can *directly* express time-based computations easily and succinctly, while allowing scalable processing on a cluster. This can enable quick building, debugging, *backtesting*, and deployment of new algorithms for such applications.

2) Real-Time-Readiness — Closing the M3 Loop

As commercial systems mature, they may wish to *close the M3 loop*, operating directly on real-time data feeds instead of performing offline computations. For example, RunningClickCount could operate over click feeds and produce an online tracker. In case of BT, the inability to operate directly on real-time data can result in missed opportunities—in Example 2, we would like to immediately detect a correlation between searches for "iCarly" and clicks on deodorant ads, and start delivering deodorant ads to such users.

Current BT schemes [7, 19, 34] suggest loading data into a distributed file system, followed by several map-reduce stages, each performing customized and often non-incremental computations on offline data. Such solutions cannot easily be converted or re-used for live data, since they are not written using an explicit temporal or incremental specification.

Another alternative is to write continuous queries deployed over a *data stream management system* (*DSMS*) [3]. This works well for real-time data, but a DSMS cannot perform massive-scale map-reduce-style offline data processing that is dominant in commercial systems today. Implementing a distributed DSMS infrastructure to process offline data at such scales is difficult, with issues such as (1) locality-based data partitioning; (2) efficient re-partitioning; (3) network-level transfer protocols; (4) system administration; and (5) the unsuitability and high overhead of low-latency process-pairs-based failure recovery schemes [4] used by real-time DSMSs. Map-reduce excels at solving these problems for offline data.

If new applications are designed specially for real-time, we incur the cost of maintaining two disparate systems as we migrate to real-time. Further, we cannot first debug, backtest, or deploy the same solutions over large-scale offline data using current map-reduce clusters, before switching to real-time.

First Contribution: We propose (§ III) a novel framework called *TiMR* (pronounced *timer*), to process temporal queries over large volumes of offline data. TiMR combines an unmodified single-node DSMS with an unmodified map-reduce distributed computing platform. Users perform analytics using a temporal language (e.g., LINQ [30] or StreamSQL). The queries run efficiently on large-scale offline temporal data in a map-reduce cluster, and are naturally real-time-ready. TiMR leverages the temporal algebra underlying the DSMS for repeatable behavior across runs. Interestingly, recent proposals to pipeline M-R [8, 23] can allow TiMR to also operate over live streams (§ VII). TiMR also incorporates several features for high performance, such as *temporal partitioning* (§ III-B) and *cost-based query fragmentation* (§ VI).

3) New Temporal Algorithms and Queries for BT

Given a framework to scalably process temporal queries, a natural question is: *are temporal queries a good choice for complex big data applications such as BT?* Further, since existing BT proposals are intimately tied to the map-reduce offline computation model, there is a need to rethink BT algorithms for every BT stage described earlier, so that they can leverage the inherent temporal nature of such computations.

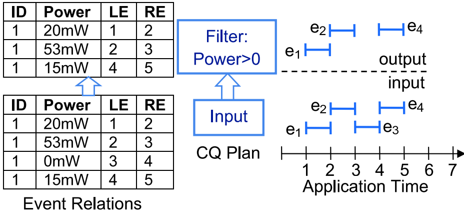

Fig. 2. Events, CQ plans, lifetimes.

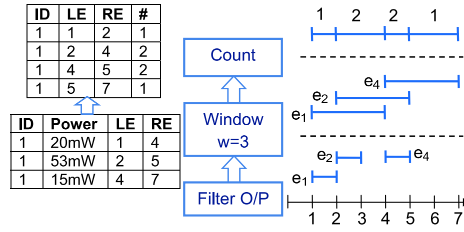

Fig. 3. Windowed count CQ plan.

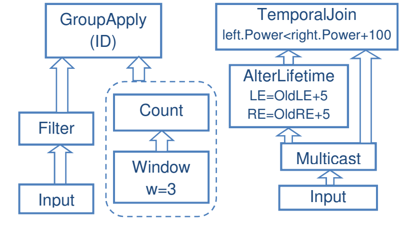

Fig. 4. GroupApply & Join plans.

Second Contribution: We propose (§ IV) a new end-to-end BT solution, with scalable temporal queries for each BT phase. Of particular interest is a novel temporal algorithm for data reduction that uses *statistical hypothesis testing* for detecting and exploiting trends such as those depicted in Example 2. While our BT solution is innovative in and of itself, it demonstrates the feasibility and generality of using temporal queries for a complex application that operates over large-scale temporal data. Our BT solution, replacing a complex custom offline-specific ad infrastructure, uses only 20 temporal queries (in LINQ) and delivers much better targeting results.

We evaluate (§ V) TiMR as well as our BT solution, using real data from our ad platform. We show that TiMR is highly scalable and efficient, and incurs orders-of-magnitude lower development cost. Further, our temporal queries for BT reduce memory requirements and learning time by up to an order-of-magnitude. They deliver better CTR (by up to several factors) and coverage than current schemes, even before taking the future potential of quicker targeting into account.

We use Microsoft StreamInsight [2] and Dryad [20] as running examples of DSMS and map-reduce, but the concepts are applicable to other DSMS and map-reduce products.

II. Background and Alternative Solutions

A. Data Stream Management Systems

A DSMS [1, 3, 17] enables applications to execute long-running *continuous queries* (*CQs*) over data streams in real time. DSMSs are used for efficient real-time processing in applications such as fraud detection, monitoring RFID readings from sensors, and algorithmic stock trading. While DSMSs target real-time data, they usually [5, 17, 22, 31] incorporate a data model, with a temporal algebra and query semantics based on early work on temporal databases [21].

A.1) Streams and Events

A *stream* is a potentially unbounded sequence $e_1, e_2, \ldots$ of events. An *event* is a notification from outside (e.g., sensor) that consists of a *payload* and a *control parameter* that provides event metadata. Two common [5, 29, 31] control parameters for events are: (1) an application-specified event generation time or *timestamp* LE, and (2) a time window [LE, RE) that indicates the period over which the event influences output. For "instantaneous" events with no lifetime, called *point events*, RE is set to LE $+\delta$ where δ is the smallest possible time-unit. A stream can be viewed as a changing "temporal relation". Figure 2 (right) shows 4 events $e_1, \ldots, e_4$, corresponding to 4 power meter readings. The corresponding temporal relation depicting events with lifetimes is also shown.

A.2) Queries and Operators

Users write CQs using languages such as StreamSQL (StreamBase and Oracle CEP) or LINQ (StreamInsight). The query is converted into a *CQ plan*, that consists of a tree of *temporal operators*, each of which performs some transformation on its input streams (*leaves*) and produces an output stream (*root*). Semantics of operators are usually defined in terms of their effect on the temporal relation corresponding to input and output streams, and are independent of when tuples are actually processed (system time). We summarize the relevant operators below; more details on operators and related issues such as time progress and state cleanup can be found in [5, 17, 22, 31].

Filter/Project Filter is a stateless operator. It selects events that satisfy certain specified conditions. For instance, the query plan in Figure 2 detects non-zero power readings (the output events and relation are also shown). Project is a stateless operator that modifies the output schema (e.g., add/remove columns or perform stateless data transformations).

Windowing and Aggregation Windowing is performed using the AlterLifetime operator, which adjusts event LE and RE; this controls the time range over which an event contributes to query computation. For window size w, we simply set RE = LE + w. This ensures that at any time t, the set of "active" events, i.e., events whose lifetimes contain t, includes all events with timestamp in the interval $(t - w, t]$.

An aggregation operator (Count, Sum, Min, etc.) computes and reports an aggregate result each time the active event set changes (i.e., every *snapshot*). Continuing the Filter example, suppose we wish to report the number of non-zero readings in the last 3 seconds. The CQ plan and events are shown in Figure 3. We use AlterLifetime to set RE = LE + 3 (we show this as a Window operator with $w = 3$ for clarity), followed by a Count operator. The CQ reports precisely the count over the last 3 secs, reported whenever the count changes.

GroupApply, Union, Multicast The GroupApply operator allows us to specify a grouping key, and a query sub-plan to be "applied" to each group. Assume there are multiple meters, and we wish to perform the same windowing count for each meter (group by ID). The CQ of Figure 4 (left) can be used to perform this computation. A related operator, Union, simply merges two streams together, while a Multicast operator is used to send one input stream to two downstream operators.

TemporalJoin and AntiSemiJoin The TemporalJoin operator allows correlation between two streams. It outputs the relational join (with a matching condition) between its left and right input events. The output lifetime is the intersection of the joining event lifetimes. Join is stateful and usually implements a *symmetric hash join* algorithm; the active events for each input are stored in a separate internal *join synopsis*. For example, the CQ in Figure 4 (right) computes time periods when the meter reading increased by more than 100 mW, compared to 5 secs back. A common application of TemporalJoin is when the left input consists of point events — in this case, TemporalJoin effectively filters out events on the left input that do not intersect any previous matching event lifetime in the right input synopsis. A related operator, AntiSemiJoin, is used to eliminate point events from the left input that do intersect some matching event in the right synopsis.

User-Defined Operators DSMSs also support incremental *user-defined operators* (*UDOs*), where the user provides code to perform computations over the (windowed) input stream.

B. The Map-Reduce (M-R) Paradigm

Many systems have embraced the map-reduce paradigm of distributed storage and processing on large clusters of shared-nothing machines over a high-bandwidth interconnect, for analyzing massive offline datasets. Example proposals include MapReduce/SawZall [10], Dryad/DryadLinq [20, 35], and Hadoop [16], where each query specifies computations on data stored in a distributed file system such as HDFS [16], GFS [14], Cosmos [6], etc. Briefly, execution in these systems consists of one or more *stages*, where each stage has two phases. The *map* phase defines the *partitioning key* (or function) to indicate how the data should be partitioned in the cluster, e.g., based on UserId. The *reduce* phase then performs the same computation (aggregation) on each data partition in parallel. The computation is specified by the user, via a *reducer method* that accepts all rows belonging the same partition, and returns result rows after performing the computation.

Under the basic model, users specify the partitioning key and the reducer method. Recently, several higher-level scripting languages such as SCOPE and Pig have emerged — they offer easier relational- and procedural-style constructs that are compiled down to multiple stages of the basic M-R model.

C. Strawman Solutions

Refer to Example 1 (RunningClickCount), which uses advertising data having the schemas depicted in Figure 1. As discussed in Section I, there are two current solutions:

- We can express it using SCOPE, Pig, DryadLinq, etc. For instance, the following SCOPE queries (note that the syntax is similar to SQL) together produce the desired output.

```
 OUT1 = SELECT a.Time, a.AdId, b.Time as prevTime
FROM ClickLog AS a INNER JOIN ClickLog AS b
WHERE a.AdId=b.AdId AND b.Time > a.Time - 6
hours;
 OUT2 = SELECT Time, AdId, COUNT(prevTime)
FROM OUT1 GROUP BY Time, AdId;
```

  Unfortunately, this query is intractable because we are performing a self equi-join of rows with the same AdId, which is prohibitively expensive. The fundamental problem is that the relational-style model is unsuitable for sequence-based processing, and trying to force its usage can result in inefficient (and sometimes intractable) M-R plans.
- A more practical alternative is to map (partition) the dataset and write our own reducers that maintain the necessary in-memory data structures to process the query. In case of RunningClickCount, we partition by AdId, and write a reducer that processes all entries in Time sequence. The reducer maintains all clicks and their timestamps in the 6-hour window in a linked list. When a new row is processed, we look up the list, delete expired rows, and output the refreshed count. This solution has several disadvantages: (1) it can be inefficient if not implemented carefully, (2) it is non-trivial to code, debug, and maintain; (3) it cannot handle deletions or disordered data without complex data structures (e.g., red-black trees), and hence requires pre-sorting of data, and (4) it is not easily reusable for other temporal queries. For comparison, the efficient implementation of aggregation and temporal join in StreamInsight consists of more than 3000 lines of high-level code each.

Further, neither of these solutions can be reused easily to directly operate over streaming data feeds.

III. The TiMR Framework

TiMR is a framework that transparently combines a map-reduce (M-R) system with a temporal DSMS. Users express time-oriented analytics using a temporal (DSMS) query language such as StreamSQL or LINQ. Streaming queries are declarative and easy to write/debug, real-time-ready, and often several orders of magnitude smaller than equivalent custom code for time-oriented applications. TiMR allows the temporal queries to transparently scale on offline temporal data in a cluster by leveraging existing M-R infrastructure.

Broadly speaking, TiMR's architecture of compiling higher level queries into M-R stages is similar to that of Pig/SCOPE. However, TiMR specializes in time-oriented queries and data, with several new features such as: (1) the use of an *unmodified* DSMS as part of compilation, parallelization, and execution; and (2) the exploitation of new temporal parallelization opportunities unique to our setting. In addition, we leverage the temporal algebra underlying the DSMS in order to guarantee repeatability across runs in TiMR within M-R (when handling failures), as well as over live data.

A. TiMR Architecture

At a high level, TiMR divides the temporal CQ plan (derived from a high-level temporal query) into partitionable subplans. Each subplan is executed as a stage in unmodified M-R, with the unmodified single-node DSMS embedded within reducers, to process rows as events. Execution on TiMR provides the benefit of leveraging the efficient DSMS sequence processing

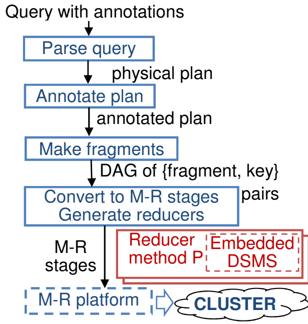

Fig. 5. TiMR architecture.

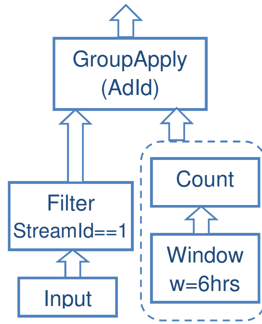

Fig. 6. CQ plan for RunningClickCount.

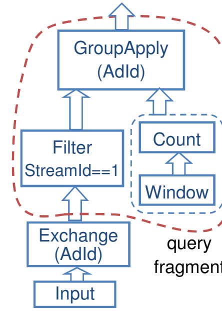

Fig. 7. Annotated CQ plan for RunningClickCount.

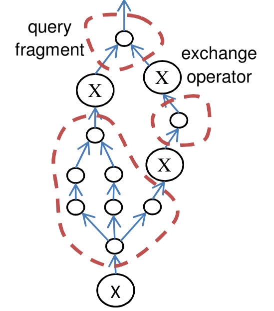

Fig. 8. Complex annotated query with three fragments.

engine, and avoids the need for customized and often complex implementations for large-scale temporal analytics.

TiMR operates in multiple steps (see Figure 5):

1) Parse Query Users write temporal queries in the DSMS language, and submit them to TiMR. This step uses the DSMS query translation component to convert the query from a high-level language into a CQ plan. The CQ plan ensures logical query correctness, but does not consider parallel execution. In our running example (RunningClickCount), the query can be written in LINQ as follows (the code for StreamSQL is similar). The corresponding CQ plan is shown in Figure 6.

```
var clickCount = from e in inputStream
  where e.StreamId == 1 // filter on some column
  group e by e.AdId into grp // group-by, then window
  from w in grp.SlidingWindow(TimeSpan.FromHours(6))
  select new Output { ClickCount = w.Count(), .. };
```

2) Annotate Plan The next step adds data-parallel execution semantics to the CQ plan. A *partition* of a stream is defined as the subset of events that reside on a single machine. A stream S_i is said to be partitioned on a set of columns X, called the *partitioning key*, if it satisfies the condition $\forall e_1, e_2 \in S_i$: $e_1[X] = e_2[X] \implies P(e_1) = P(e_2)$, where $P(e)$ denotes the partition (or machine) assigned to event e and $e[X]$ denotes the corresponding subset of column values in event e.

Parallel semantics are added to the CQ plan by inserting logical *exchange operators* into the plan. An exchange operator E_X logically indicates a repartitioning of the stream to key X. We can annotate a CQ plan by allowing the query writer to provide explicit annotations in the form of *hints*, along with query specification. Alternatively, we can build an optimizer to choose the "best" annotated plan for a given CQ. This is a non-trivial problem, as the following example illustrates.

Example 3. *Consider a CQ plan with two operators, O_1 followed by O_2. Assume O_1 has key $\{UserId, Keyword\}$, while O_2 has key $\{UserId\}$. A naive annotation would partition by $\{UserId, Keyword\}$ before O_1 and repartition by $\{UserId\}$ before O_2. However, based on statistics such as repartitioning cost, we may instead partition just once by $\{UserId\}$, since this partitioning implies a partitioning by $\{UserId, Keyword\}$. We encountered this scenario in BT and found the latter choice to be* $2.27\times$ *faster for a real dataset (Sections IV and V).*

Individual operator semantics and functional dependencies in the data govern the space of valid annotated plans. For example, a GroupApply with key X can be partitioned by X or any subset of X. The details of parallelization opportunities presented by operators, and how we can leverage database query optimization [15, 36] to find a low-cost annotated CQ plan, are covered in Section VI. In Section III-B, we cover a new form of *temporal parallelism*, that exploits bounded windows to partition data using the time attribute.

Figure 7 shows an annotated CQ plan for RunningClickCount. This simple plan contains one exchange operator that partitions the stream by {AdId}; this plan is valid since RunningClickCount performs GroupApply with key AdId.

3) Make Fragments This step converts the annotated plan into a series of *computation stages* as follows. Starting from the root, it performs a top-down traversal of the tree and stops when it encounters an exchange operator along all paths. The query subplan encountered during the traversal is called a *query fragment*, and the fragment is parallelizable by the partitioning set of the encountered exchange operators[1]; this set is referred to as the partitioning key of the fragment. The traversal process is repeated, generating further {fragment, key} pairs until we reach the leaves of the CQ plan. In our running example, we create a single fragment consisting of the original RunningClickCount CQ plan, with AdId as key. A more complex annotated plan outline is shown in Figure 8, along with the query fragments.

4) Convert to M-R The final step converts the set of {fragment, key} pairs into a corresponding set of M-R stages, as follows. Assume that each fragment has one input and one output stream (this assumption is relaxed in Section III-C). For each fragment, TiMR creates a M-R stage that partitions (maps) its input dataset by the partitioning key. M-R invokes a stand-alone reducer method P for each partition in parallel. P is constructed by TiMR using the query fragment. P reads rows of data from the partition (via M-R), and converts each row

[1]These partitioning sets are guaranteed to be identical, since multi-input operator such as TemporalJoin have identically partitioned input streams.

into an event using a predefined Time column[2]. Specifically, it sets event lifetime to $[\text{Time}, \text{Time} + \delta)$ (point event) and the payload to the remaining columns. P then passes these events to the *original unmodified DSMS* via a generated method P'. P' is an embedded method that can execute the original CQ fragment using a DSMS server instance created and embedded in-process. The DSMS performs highly efficient in-memory event processing within P' and returns query result events to P, which converts the events back into rows that are finally passed back to M-R as the reducer output.

In our running example, TiMR sets the partitioning key to AdId, and generates a stand-alone reducer P that reads all rows (for a particular AdId), converts them into events, and processes the events with the above CQ, using the embedded DSMS. Result events are converted back into rows by TiMR and returned to M-R as reducer output.

B. Temporal Partitioning

Many CQs (e.g., RunningClickCount for a single ad) may not be partitionable by any data column. However, if the CQ uses a window of width w, we can partition computation based on time as follows. We divide the time axis into overlapping *spans* $S_0, S_1, \ldots$, such that the overlap between successive spans is w. Each span is responsible for output during a time interval of width s, called the *span width*. Let t denote a constant reference timestamp. Span S_i receives events with timestamp in the interval $[t+s \cdot i - w, t+s \cdot i + s)$, and produces output for the interval $[t + s \cdot i, t + s \cdot i + s)$. Note that some events at the boundary between spans may belong to multiple partitions.

The overlap between spans S_{i-1} and S_i ensures that the span S_i can produce correct output at time $t + s \cdot i$; this is possible only if a window w of events is available at that time, i.e., S_i receives events with timestamps from $t + s \cdot i - w$ onwards. In case of multiple input streams in a fragment, the span overlap is the maximum w across the streams. A greater span width s (relative to w) can limit redundant computation at the overlap regions, at the expense of fewer data partitions. Temporal partitioning can be very useful in practice, since it allows scaling out queries that may not be otherwise partitionable using any payload key. In Section V, we will see that a sliding window aggregate query without any partitioning key, gets a speedup of 18× using temporal partitioning.

C. Discussion

Note that we do not modify either M-R or the DSMS in order to implement TiMR. TiMR works independently and provides the plumbing necessary to interface these systems for large-scale temporal analytics. From M-R's perspective, the method P is just another reducer, while the DSMS is unaware that it is being fed data from the file system via M-R. This feature makes TiMR particularly attractive for use in conjunction with commercial DSMS and map-reduce products. We discuss some important aspects of TiMR below.

1) Online vs. Offline The use of a real-time DSMS for offline data is possible because of the well-defined temporal algebra upon which the DSMS is founded. The DSMS only uses *application time* [31] for computations, i.e., timestamps are a part of the schema, and the underlying temporal algebra ensures that query results are independent of when tuples physically get processed (i.e., whether it runs on offline or real-time data). This aspect also allows TiMR to work well with M-R's failure handling strategy of restarting failed reducers—the newly generated output is guaranteed to be identical when we re-process the same input partition.

It is important to note that TiMR enables temporal queries on *large-scale offline data*, and does not itself attempt to produce low-latency real-time results for real-time streams (as an aside, TiMR can benefit from pipelined M-R; cf. Section VII). The queries of course are ready for real-time execution on a DSMS. Conversely, real-time DSMS queries can easily be back-tested and fine-tuned on large-scale offline datasets using TiMR.

2) Push vs. Pull One complication is that the map-reduce model expects results to be synchronously returned back from the reducer, whereas a DSMS pushes data asynchronously whenever new result rows get generated. TiMR handles this inconsistency as follows: DSMS output is written to an in-memory *blocking queue*, from which P reads events synchronously and returns rows to M-R. Thus, M-R blocks waiting for new tuples from the reducer if it tries to read a result tuple before it is produced by the DSMS.

3) Partitioning M-R invokes the reducer method P for each partition; thus, we instantiate a new DSMS instance (within P) for every AdId in RunningClickCount, which can be expensive. We solve this problem by setting the partitioning key to hash(AdId) instead of AdId, where hash returns a hash bucket in the range [1...#machines]. Since the CQ itself performs a GroupApply on AdId, output correctness is preserved.

4) Multiple Inputs and Outputs While the vanilla M-R model has one logical input and output, current implementations [6, 25] allow a job to process and produce multiple files. In this case, fragments with multiple inputs and/or outputs (see Figure 8) can be directly converted into M-R stages.

We can support the vanilla M-R model by performing an automated transformation for CQ fragments and intermediate data. Briefly, we union the k inputs into a common schema with an extra column C to identify the original source, before feeding the reducer. Within the CQ, we add a multicast with one input and k outputs, where each output selects a particular source (by filtering on column C) and performs Project to get back the original schema for that stream. A similar transformation is done in case a fragment produces multiple outputs.

In case of BT, we can avoid the above transformation step for input data as follows. The BT streams of Figure 1 are instead directly collected and stored using the unified schema of Figure 9. Here, we use StreamId to disambiguate between

[2]The first column in source, intermediate, and output data files is constrained to be Time (i.e., the timestamp of activity occurrence), in order for TiMR to transparently derive and maintain temporal information. The extension to interval events is straightforward.

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

Fig. 9. Unified schema for BT data.

the various sources. Specifically, 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. BT queries are written to target the new schema, and thus operate on a single input data source.

We implemented TiMR with StreamInsight and SCOPE/Dryad to run queries over large-scale advertising data, and found it to be scalable and easy-to-use. We now switch gears, and focus on a real-world validation of TiMR's value by proposing new temporal-aware BT techniques and showing how TiMR can implement these techniques efficiently and with minimal effort.

IV. BT Algorithms with TiMR

Recall that BT uses information collected about users' online behavior (such as Web searches and pages visited) in order to select which ad should be displayed to that user. The usual goal of BT is to improve CTR by showing the most relevant ad to each user based on an analysis of historical behavior.

A. Overview

We refer to observed user behavior indicators such as search keywords, URLs visited, etc. as *features*. For simplicity, we will use the terms feature and keyword interchangeably in this paper. We next formally define the concept of *user behavior profile* (*UBP*) [7, 34] which basically represents user behavior in the Bag of Words model [28], where each word is a feature.

Definition 1 (Ideal UBP)**.** *The ideal user behavior profile for each user U_i at time t and over a historical time window parameter of τ (time units), is a real-valued array $\bar{U}_i^t = \langle U_{i,1}^t, \ldots \rangle$ with one dimension for each feature (such as search keyword or URL). The value $U_{i,j}^t$ represents the weight assigned to dimension j for user U_i, as computed using their behavior over the time interval $[t - \tau, t)$.*

A common value assigned to $U_{i,j}^t$ is simply the number of times that user U_i searched for term j (or visited the webpage, if j is a URL) in the time interval $[t-\tau, t)$. Common variations include giving greater importance to more recent activities [7] by using a weighting factor as part of the weight computation. For each ad, we collect prior ad impression/click information associated with user behavior profiles $\mathcal{D}$ that consists of n *observations* (corresponding to n impressions of this ad), $\mathcal{D} = \{(\bar{x}_1, y_1), \ldots, (\bar{x}_n, y_n)\}$, where $\bar{x}_k$ is the behavior profile of the user at the time when she is displayed with this ad, and y_k indicates whether she clicked ($y_k = 1$) this ad or not ($y_k = 0$).

The core insight behind our BT approach is that ad click likelihood depends only on the UBP at the time of the ad presentation. Based on this insight, our ideal goal is to accurately estimate (for each ad) the expected CTR given a UBP $\bar{U}_i^{t'}$ at any future time t'.

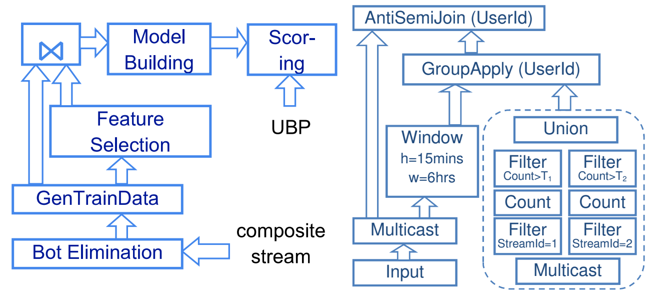

Fig. 10. BT architecture. Fig. 11. Bot elimination.

Practical Restrictions In most commercial platforms, the ideal UBPs are prohibitively large, with billion of users and millions of keywords/URLs. Thus, we need effective feature selection techniques to make subsequent CTR estimation tractable and accurate. In case of parameter τ, commercial systems consider relatively long-term user behavior (e.g., weeks) while other prefer short-term behavior (less than a day). Prior work [34] over real data has shown that short-term BT can significantly outperform long-term BT. Based on this finding, we use τ = 6 hours is this work.

Note that it is not feasible to build an estimator for every ad. We need to group ads into ad classes and build one estimator for each class. One solution is to group ads manually (e.g., "electronics", "games"). A better alternative is to derive data-driven ad classes, by grouping ads based on the similarity of users who click (or reject) the ad. We do not focus on these choices, but use the term ad to generally refer to ad classes.

B. System Architecture for BT

We now present temporal algorithms for end-to-end BT, starting with data having the schema in Figure 9. We show CQ plans broken into separate steps for readability.

Figure 10 shows our BT architecture. We first get rid of bots, and then generate per-impression UBPs, which are associated with labels (indicating impression being clicked or not) to compose training data for feature selection. The reduced training data (with selected features) are fed to a model builder, which produces a model to predict expected CTR for each ad, i.e., the likelihood for the ad to be clicked. The models are used to score incoming UBPs in real-time, predict CTR for each ad, and hence choose the ad with the maximum expected benefit (including CTR and other factors) to be delivered. To ease presentation, we describe the steps of generating training data, feature selection, and model building and scoring for a given ad in the following sections.

B.1) Bot Elimination

We first get rid of 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. In a one week dataset, we found that 0.5% of users are classified as bots using a threshold of 100, but these users contribute to 13% of overall clicks and

searches. Thus, 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 will be diluted by the spurious behavior of bots.

Implementation The *BotElim* CQ shown in Figure 11 gets rid of bots. We first create a hopping window (implemented using the AlterLifetime operator from Section II-A) with hop size $h = 15$ mins and window size $w = 6$ hours, over the original composite source S_1. This updates the bot list every 15 mins using data from a 6 hour window. The GroupApply (with grouping key UserId) applies the following sub-query to each UserId sub-stream. From the input stream for that user, we extract the click and keyword data separately (by filtering on StreamId), 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 S_2 that retains only bot users' data. We finally perform an AntiSemiJoin (on UserId) of the original point event stream S_1 with S_2 to output non-bot users' data. Note that UserId serves as the partitioning key for BotElim.

B.2) Generating Training Data

This component (*GenTrainData*) is used by feature selection, model building, and scoring. The goal is to generate *training data* of positive and negative examples in the form $\langle \bar{x}$ = UBP, y = click or not$\rangle$. Positive examples include the ad impressions being clicked (i.e., $y = 1$); negative examples include the ad impressions not being clicked (i.e., $y = 0$); and UBP is the user behavior profile defined at the time of the ad impression.

We first detect *non-clicks* (ad impressions that do not result in a click) by eliminating impressions that are followed by a click (by the same user) within a short time d (clicks are directly available as input). We also generate per-user UBPs based on user searches and pageviews. Finally, whenever there is an click/non-click activity for a user, a training example is generated by joining the activity with that user's UBP.

Implementation Refer to Figure 12. We first get rid of impressions that resulted in a click, by performing an AntiSemiJoin of impression point events with click data whose LE is moved $d = 5$ minutes into the past. We call the resulting composite stream of clicks and non-clicks S_1. We extract the keyword stream from the input using a filter on StreamId, and then perform a GroupApply by {UserId, Keyword}: for each substream, we perform windowing ($w = \tau$) followed by Count, to produce a stream S_2 of {UserId, Keyword, Count}, where Count is the number of times Keyword was used by UserId in the last 6 hours. Note that this is exactly the UBPs (in sparse representation) refreshed each time there is user activity. Finally, we perform a TemporalJoin (on UserId) between S_1 and S_2 to output, for each click and non-click, the associated UBP in sparse format.

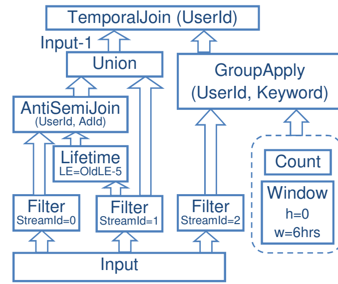

Fig. 12. GenTrainData

GenTrainData scales well using UserId as partitioning key. Note that we could instead have partitioned by {UserId, Keyword} for generating UBPs, but this is not useful since (1) there is already a large number of users for effective parallelization, and (2) we anyway need to partition by UserId alone for the subsequent TemporalJoin. This optimization scenario was depicted in Example 3, and is evaluated later.

B.3) Feature Selection

In the training data of $\langle \bar{x}$ = UBP, y = click or not$\rangle$, the ideal UBPs have very large dimensionality; in our experiments we have over 50M distinct keywords as original dimensions. It is impractical and meaningless to perform data analysis in such high-dimensional space, since most of these features are not useful to predict CTR of an ad, and it unnecessarily requires too much training data for model learning. We thus need to do feature selection for both computational feasibility and accuracy of model learning and scoring.

Several scalable techniques have been proposed for feature selection in BT. For example, we can retain the most popular keywords [7]. However, this may retain some common search keywords (e.g., "facebook", "craigslist", etc.) that may not be good predictors for ad click or non-click. Another alternative is to map keywords into a smaller domain of categories in a *concept hierarchy* such as ODP [27] (e.g., electronics, fitness, etc.). However, this technique cannot adapt to new keywords and user interest variations. Further, the manual update of the concept hierarchy introduces delays and inaccuracies.

We propose *keyword elimination* based on *statistical hypothesis testing* [11]. The basic intuition is that we want to retain any keyword that we can establish with confidence to be positively (or negatively) correlated with ad clicks, based on the relative frequency of clicks with that keyword in the UBP (compared to clicks without that keyword).

We first get rid of keywords without sufficient *support* to make the hypothesis testing *sound*; we define support as the number of times that a keyword appears in the UBPs associated with ad clicks (not ad impressions without click). Next, we use the unpooled two-proportion z-test [11] to derive a score for each keyword, that is representative of the relevance of that keyword to the ad. Highly positive (or negative) scores indicate a positive (or negative) correlation to ad clicks. We can then place a threshold on the absolute score to retain only those keywords that are relevant to the ad in a positive or negative manner. Let C_K and I_K denote the number of clicks and impressions respectively, for a particular ad and with keyword K in the user's UBP at the time of impression occurrence. Further, let $C_{\bar{K}}$ and $I_{\bar{K}}$ denote total clicks and impressions respectively for the ad, without keyword K. We have the CTR (probability of clicking) with keyword $p_K = C_K/I_K$ and the CTR without keyword $p_{\bar{K}} = C_{\bar{K}}/I_{\bar{K}}$. The intuition is for a

given keyword, if there is no significant difference between the two CTRs with and without the keyword in UBFs at ad impressions, it means this keyword is not relevant to the ad. Formally, the null hypothesis for the statistical testing is { H_0: keyword K is independent of clicks on the ad }. We can compute the z-score for this hypothesis as:

$$z = \frac{p_K - p_{\bar{K}}}{\sqrt{\frac{p_K(1-p_K)}{I_K} + \frac{p_{\bar{K}}(1-p_{\bar{K}})}{I_{\bar{K}}}}}$$

given that we have at least 5 independent observations of clicks and impressions with and without keyword K. The z-score follows the $N(0,1)$ Gaussian distribution if H_0 holds. Hence, at 95% confidence level, if $|z| > 1.96$, we will reject hypothesis H_0, and thus retain keyword K. An appropriate threshold for $|z|$ enables effective data-driven keyword elimination.

Implementation As shown in Figure 13, we first compute the total number of non-clicks and clicks using GroupApply (by AdId) followed by Count with h covering the time interval over which we perform keyword elimination. This sub-query (called *TotalCount*) is partitioned by AdId. Next, we process the output of GenTrainData to compute the number of non-clicks and clicks with each keyword K. In this sub-query (called *PerKWCount*), the partitioning key is {AdId, Keyword}.

These two streams are joined to produce a stream with one tuple for each {AdId, Keyword} pair, that contains all the information to compute the z-test. We compute the z-score using a UDO (cf. Section II-A), and a filter eliminates keywords whose z-scores fall below a specified threshold. This sub-query, called *CalcScore*, uses {AdId, Keyword} as partitioning key. Finally, we perform a TemporalJoin (not shown) of the original training data with the reduced keyword stream to produce the reduced training data.

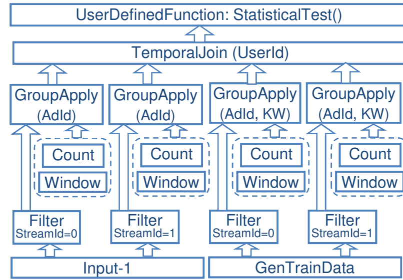

Fig. 13. Feature selection.

B.4) Model Generation and Scoring

We have described how to use feature selection to generate reduced training data of the form $\langle \bar{x} = \text{UBP}, y = \text{click or not} \rangle$ for each ad impressions. We use the training data to learn a logistic regression (LR) model as $y = f(\bar{x}) = \frac{1}{1+e^{-(w_0+\bar{w}^T\bar{x})}}$, where $\bar{w}$ is called the weight vector and f is the logistic function [11]. For a user with UBP$_i$, the model can predict the probability $f(\text{UBP}_i)$ that this user will click the ad if it is shown to her. Note that the predicted value $f(\text{UBP}_i)$ is between 0 and 1, measuring the probability of click. We choose LR because of its simplicity, good performance, and fast convergence.

One challenge with model learning is that the input data is highly skewed with mostly negative examples, as the CTR is typically lower than 1%. Hence, we create a balanced dataset by sampling the negative examples. This implies that the LR prediction y is no longer the expected CTR, whereas we need CTR to compare predictions across ads. We can estimate the CTR for a given prediction y as follows: we compute predictions for a separate validation dataset, choose the k nearest validation examples with predictions closest to y, and estimate CTR as the fraction of positive examples in this set.

When we have the opportunity to serve a user with an ad, the user's UBP is scored to get a list of ads ranked by CTR. The ad delivery engine uses this list to choose an ad placement.

Implementation Model generation is partitionable by AdId. Our input (for each ad) is a stream of ⟨UBP, outcome⟩ examples. We use a UDO (cf. Section II-A) with window, to perform in-memory LR on this data. The hop size determines the frequency of performing LR, while window size determines the amount of training data used for learning. The output model weights are lodged in the right synopsis of a TemporalJoin operator (for scoring), so we can generate a prediction whenever a new UBP is fed on its left input.

Our BT algorithms are fully incremental, using stream operators. We can plug-in an incremental LR algorithm, but given the speed of LR convergence (due to effective data reduction), we find periodic recomputation of the LR model, using a UDO over a hopping window, to work well for BT.

V. Evaluation

A. Setup and Implementation

We use a dataset consisting of one week's worth of logs (several terabytes) collected from our ad platform. The logs consist of clicks, impressions, and keyword searches, and have the composite schema shown in Figure 9. We split the dataset into training data and test data equally. We consider the 10 most popular ad classes in our experiments. Note that real-time event feeds would also naturally follow the same schema. There are around 250M unique users and 50M keywords in the log. Our experimental cluster consists of around 150 dual core 2GHz machines with 8 GB RAM, and uses Cosmos [6] as the distributed file system.

We implemented the TiMR framework (see Section III) to work with StreamInsight and SCOPE/Dryad. We then used TiMR for BT, by implementing the solutions proposed in Section IV. In order to compare against TiMR, we also implemented our BT solution using hand-written C# reducers.

B. Evaluating TiMR

Implementation Cost We use lines (semicolons) of code as a proxy for development effort. Figure 14(left) shows that end-to-end BT using TiMR uses 20 easy-to-write temporal queries in LINQ, compared to 360 lines with custom reducers. The custom solution was more difficult to debug and optimize (several weeks of effort), makes multiple passes over the data, and cannot be used directly with real-time event feeds.

Performance Our first reducer implementations were too slow due to subtle data structure inefficiencies. We carefully optimized the reducer and found the runtime to be around 3.73 hours for end-to-end BT over the log. In comparison,

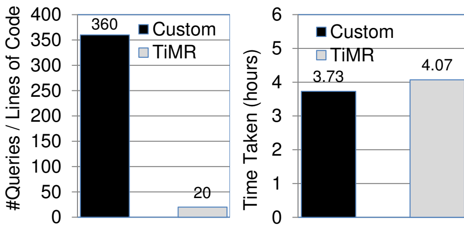

Fig. 14. Lines of code & processing time.

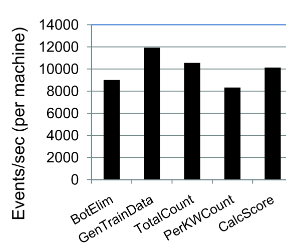

Fig. 15. Throughput.

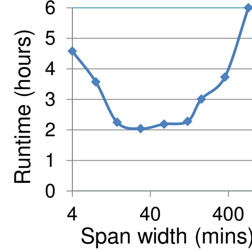

Fig. 16. Time-partitioning.

Highly Positive		Highly Negative	
Keyword	Score	Keyword	Score
celebrity	11.0	verizon	−1.3
icarly	6.7	construct	−1.4
tattoo	8.0	service	−1.5
games	6.5	ford	−1.6
chat	6.5	hotels	−1.8
videos	6.4	jobless	−1.9
hannah	5.4	pilot	−3.1
exam	5.1	credit	−3.6
music	3.3	craigslist	−4.4

Fig. 17. Keywords, deodorant ad.

Highly Positive		Highly Negative	
Keyword	Score	Keyword	Score
dell	28.6	pregnant	−2.9
laptops	22.8	stars	−4.0
computers	22.8	wang	−4.2
Juris	21.5	vera	−4.2
toshiba	12.7	dancing	−4.2
vostro	12.6	myspace	−8.0
hp	9.1	facebook	−8.6

Fig. 18. Keywords, laptop ad.

Highly Positive		Highly Negative	
Keyword	Score	Keyword	Score
blackberry	27.5	recipes	−1.35
curve	19.8	times	−1.54
enable	17.1	national	−1.58
tmobile	15.8	hotels	−1.69
phones	15.4	people	−1.79
wireless	15.3	baseball	−1.85
att	13.7	porn	−2.33
verizon	12.9	myspace	−2.81

Fig. 19. Keywords, cellphone ad.

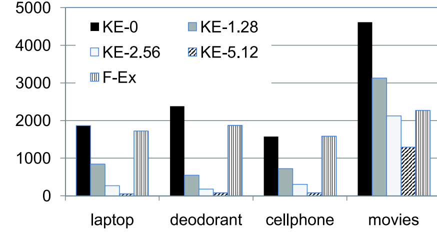

Fig. 20. Dimensionality reduction.

TiMR took 4.07 hours (see Figure 14(right)). TiMR pays a low performance overhead (< 10%), while using more general and re-usable operators compared to hand-optimized reducers. Note that this is a small price to pay for the benefits outlined in Section I. Figure 14 also shows that TiMR takes only around 4 hours to process one week of logs. Thus, when we use the same temporal queries in a real-time setting, a similarly scaled-out DSMS (or MapReduce Online [8]) should easily handle our queries. Figure 15 shows *per-machine* DSMS event rates for each of the various BT sub-queries defined in Section IV-B. Since all the queries are partitionable, performance scaled well with the number of machines.

Fragment Optimization We ran two alternate annotated CQ plans for GenTrainData (cf. Section IV-B). The first plan naively partitions UBP generation by {UserId, Keyword}, and the rest of the plan by {UserId}. The second plan (one that would be chosen by an optimizer) generates a single fragment partitioned by {UserId}. The latter plan takes 1.35 hours, against 3.06 hours for the first plan — a 2.27× speedup.

Temporal Partitioning We run a 30-minute sliding window count query over our dataset. This query is only partitionable by Time. Figure 16 shows that small span widths perform badly, mainly due to duplication of work at boundaries (overlap). On the other hand, large span widths do not provide enough parallelism. The optimal span width (∼ 30-60 mins for this dataset) is around 18× faster than single-node execution.

C. Evaluating Data Reduction

We evaluate the following data reduction techniques:

- *KE-z*: Keyword elimination with z-score threshold set to z. We vary z from 0 to 5.12. Note that $z = 0$ implies that we retain all keywords with sufficient support (at least 5 clicks exist with that keyword, across the UBPs of all users).
- *F-Ex*: This alternative, currently used in production, performs feature extraction using a content categorization engine to map keywords to one or more predefined *categories* (each with a confidence level). The categories are based on a pre-defined concept hierarchy similar to ODP [27].
- *KE-pop*: This feature selection scheme used by Chen et al. [7] retains the most popular keywords in terms of total ad clicks or rejects with that keyword in the user history.

Relevance of Retained Keywords We found that on average, there were around 50 million unique keywords in the source data; this leads us to the question of whether our feature selection technique can produce meaningful results. Figures 17, 18, and 19 show a snapshot of keywords and their z-scores for the deodorant, laptop, and cellphone ad classes.

In case of the deodorant ad, we notice that the teen demographic is more interested in the product, based on keywords such as "celebrity", "hannah", "exam", and "icarly"[3]. On the other hand, it appears that people searching for "job" or "credit" are unlikely to click on a deodorant ad. Interestingly, users searching for "Vera Wang" or "Dancing with the Stars" are unlikely to click on laptop ads. We note that correlations exist in user behavior; these may change rapidly based on current trends and are difficult to track using static directory-based feature extraction. Further, frequency-based feature selection cannot select the best keywords for BT, as it retains common words such as "google", "facebook", and "msn", which were found to be irrelevant to ad clicks.

Dimensionality Reduction Figure 20 shows the number of keywords remaining with KE-z, for various z-score thresholds. Note that only retaining keywords with sufficient support ($z = 0$) immediately reduces the number of keywords for each ad dramatically. For comparison, we also show the number of dimensions retained after feature extraction using F-Ex; this number is always around 2000 due to the static mapping to a pre-defined concept hierarchy. KE-pop is not shown since the

[3] Hannah Montana and iCarly are popular TV shows targeted at teenagers.

Examples Chosen	laptop ad class			cellphone ad class		
	#click	#impr	CTR Lift (%)	#click	#impr	CTR Lift (%)
All	8400	3180K	0	3332	436K	0
≥ 1 pos kw	2076	398K	100	2408	237K	33
≥ 1 neg kw	7376	3019K	−8	1491	277K	−28
Only pos kws	1024	161K	146	1841	159K	53
Only neg kws	6324	2782K	−15	924	199K	−39

Fig. 21. Keyword elimination and CTR.

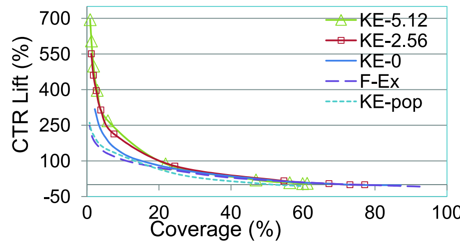

Fig. 22. CTR vs. coverage (movies).

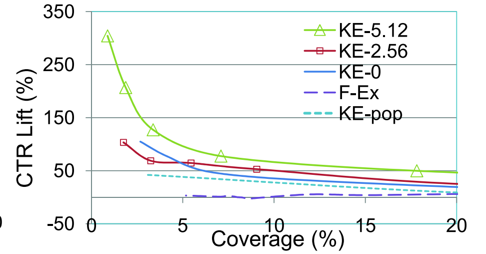

Fig. 23. CTR vs. coverage (dieting).

keywords remaining can be adjusted arbitrarily by varying the popularity threshold. We note that our technique can reduce the number of keywords retained, by up to an order of magnitude, depending on the threshold chosen.

Impact on CTR We show the effect of keywords with highly positive and negative scores on CTR. On the test data, we first compute the overall CTR (i.e., #clicks/#impressions) for an ad, denoted as V_0. We use keyword elimination to generate keywords with $|z| > 1.28$ (80% confidence level). We then create four example sets from the test data, having: (1) UPBs with at least one positive-score keyword, (2) UBPs with at least one negative-score keyword, (3) UBPs with only positive keywords; and (4) UBPs with only negative keywords. On each set, we compute the new CTR V'. The impact on CTR is then measured by *CTR lift*, defined as $V' - V_0$. Figure 21 shows that we get significant CTR Lift using examples with positive keywords, while negative examples have a negative lift (lift is only slightly negative because there are many more negative examples). Thus, keywords are a good indicator of CTR.

D. Evaluating End-To-End BT

We evaluate the effectiveness of our BT solution as follows. We build an LR model with the reduced training data, and evaluate it on the test data after applying the same feature selection learned in the training phase. The LR model produces, for each ad impression in the test data, a value y between 0 and 1 to predict whether the ad will be clicked. If the model learning is effective, we expect the impression with high prediction y to actually result in a click. We thus set a threshold on y and compute the CTR V' on those examples whose predicted value y is above this threshold; we define *coverage* as the percentage of such examples among the test data. In an extreme case, if the threshold is set to 0, all the examples in test data satisfy this threshold; thus the computed CTR is the overall CTR V_0 (CTR lift $= V' - V_0$ is 0), while the coverage is 1. Clearly, there is a tradeoff between CTR lift and coverage, depending on the threshold. Hence, we vary the threshold and evaluate BT using a plot of CTR Lift vs. coverage. The bigger the area under this plot, the more effective the advertising strategy.

CTR Lift vs. Coverage Figures 22 and 23 show the CTR Lift (varying coverage) for the "movies" and "dieting" ad classes. We see that KE-z schemes perform very well, and result in several times better CTR Lift as compared to F-Ex and KE-pop, at between 0 to 20% coverage. Interestingly, KE-pop does not perform as well as KE-z because it does not take the correlation of keywords to ad clicks into account. Note that lower coverage levels are very important, because there are typically hundreds of ad classes to choose from, and hence, by selecting ads with higher expected CTR for each impression opportunity, we can improve the overall CTR significantly. Further, we note from Figure 22 that increasing the $|z|$ threshold has an acceptable impact on maximum coverage.

Algorithm 1: Top-down Cascades-based optimizer.

```
1  OptimizePlan(Plan p, Properties requiredProps) begin
2      list validPlans ← φ;
3      list l ← PhysicalTransform(p.root, requiredProps);
4      foreach Transformation t in l do
5          Properties newProps ← GetChildRequiredProperties(t);
           // Recursively optimize each child of t
6          Plan childPlan ← OptimizePlan(t.child, newProps);
7          Properties deliveredProps ← GetDeliveredProperties(childPlan);
8          if PropertyMatch(newProps, deliveredProps) then
9              validPlans.Enqueue(new Plan(t, childPlan));
10     return CheapestPlan(validPlans);
11 end
```

Memory and Learning Time We compare memory utilization in terms of the average number of entries in the sparse representation for the UBPs, across all examples in the training data. For the "laptop" ad class, the average user vector size without data reduction is 3.7 entries per user (for that ad class), and KE-1.28 drops the size to 0.65. In comparison, F-Ex results in an average of 8 entries per UBP, since each keyword potentially maps to 3 categories. LR learning time for the "diet" ad is around 31, 18, and 5 seconds for F-Ex, KE-1.28, and KE-2.56 respectively; F-Ex takes longest due to the larger dimensionality.

VI. Extension: Optimizing CQ Plan Annotation

Recall from Section III that we wish to add exchange operators to the CQ plan. A stream S_i is said to be partitioned on a set of columns X, called the *partitioning key*, if it satisfies the condition $\forall e_1, e_2 \in S_i : e_1[X] = e_2[X] \implies P(e_1) = P(e_2)$, where $P(e)$ denotes the partition (or machine) assigned to event e and $e[X]$ denotes the corresponding subset of column values in event e. A special partitioning key $\perp$ denotes a randomly partitioned stream, while $\emptyset$ denotes a non-partitioned stream.

We can adapt a transformation-based top-down optimizer such as Cascades [15] (used in SQL Server), to annotate the CQ plan. SCOPE recently used Cascades to incorporate partitioning in set-oriented queries [36]. During physical exploration (cf. Algorithm 1, which is based on Cascades),

starting from the output, each operator recursively invokes the optimizer for each of its child plans (Line 6). The invocation includes *required properties* that need to be satisfied by the chosen child plan. In our setting, properties are defined as set of partitioning keys that the parent can accept.

During optimization of a subplan with root operator O, we can transform O (Line 3) as follows. Let $\bar{X}$ denote the set of valid partitioning keys that operator O can accept, and that are compatible with the requirement from the invoking operator (we will shortly discuss techniques to derive this information). There are two alternatives: (1) for each partitioning key X in $\bar{X}$, we add an exchange operator with key X below O and recursively optimize O's child plan with no property requirements; (2) we do not add an exchange operator, but instead recursively optimize the child plan with partitioning requirement $\bar{X}$. During optimization, we take care not to consider plans whose delivered properties are incompatible with the requirements of the invoking operator (Line 8). The end result of the optimization process is an annotated CQ plan.

Deriving Required Properties for CQ Operators Partitioning on set X or $\{Time\}$ implies a partitioning on $\perp$ as well as any set $\mathcal{Y} \supset X$. We can thus identify, for each operator, the partitioning requirements on its input:

- Consider a GroupApply sub-plan that groups by key X, or a TemporalJoin or AntiSemiJoin with an equality condition on attributes X. These operators require their input streams to be partitioned by any set of keys $\mathcal{P}$ s.t. $\emptyset \subseteq \mathcal{P} \subseteq X$.
- Multi-input operators require both incoming streams to have the same partitioning set.
- Stateless operators such as Select, Project, and AlterLifetime can be partitioned by any subset of columns, as well as by $\perp$ (i.e., they impose no specific requirements).
- Any operator with a windowed input stream can be partitioned by Time (see Section III-C).

We can also use functional dependencies and column equivalences to identify further valid partitioning keys [36]. For example, if $(X - \{C\}) \rightarrow C$, i.e., columns $(X - \{C\})$ functionally determine column C, then partitioning X implies the partitioning $X - \{C\}$.

Cost Estimation The optimizer has to estimate the cost of an annotated subplan. An exchange operator is associated with the cost of writing tuples to disk, repartitioning over the network, and reading tuples after repartitioning. The cost of DSMS operators can be estimated as in prior work [33]; if an operator executes over a partitioned stream, its cost is reduced, based on the number of machines and estimated partitions.

VII. Related Work

MapReduce Variants M-R has recently gained significant interest. Proposals such as Pig and SCOPE compile higher-level code into M-R stages. While we also compile queries into M-R, we focus on temporal data and queries, with several unique features: (1) suggesting a temporal language as the user programming surface; (2) leveraging temporal algebra to guarantee identical results on offline and real-time data; (3) using an unmodified DSMS as part of compilation, parallelization, and execution; and (4) exploiting automatic optimization and temporal parallelism opportunities in the queries.

Pig can incrementally push data to external executables [13] as part of the data processing pipeline. MapReduce Online [8] (MRO) and SOPA [23] allow efficient data pipelining in M-R across stages, but require modification to M-R. These proposals have a different goal from our work, that of changing M-R to handle real time data efficiently. We propose a declarative temporal programming model for large-scale queries, which is easy to program to and allows cost-based optimization into M-R stages for offline logs. We can transparently take advantage of the above proposals, if available in the underlying M-R platform, to (1) improve execution efficiency and latency for offline data; and (2) directly support real-time CQ processing at scale. S4 [26] supports scalable real-time processing with a M-R-style API, but does not target declarative specification or reuse of existing DSMS and M-R platforms. Recent efforts [18, 25] make periodic computations over M-R efficient by reusing prior work; this research is quite different from our goal of allowing temporal queries over offline data in M-R, while being real-time-ready. Further, we also propose new temporal algorithms to improve Web advertising with BT.

BT Many BT techniques for advertising data have been proposed. Hu et al. [19] use BT schemes to predict users' gender and age from their browsing behavior. Chen et al. [7] propose scalable BT techniques for cluster-based scenarios, while Yan et al. [34] explore the value of BT in improving CTR. These techniques are geared towards non-incremental offline data analysis. In contrast, our techniques use easy-to-specify and real-time-ready temporal queries, and TiMR allows the execution of such temporal queries over M-R. We propose incremental feature selection based on statistical testing, to determine which keywords to retain. In order to strengthen the signal, feature selection could be preceded by keyword clustering, using techniques such as Porter Stemming [32].

VIII. Conclusions

The *temporal-analytics-temporal-data* characteristic is observed for many "big data" applications such as behavioral targeted Web advertising, network log querying, and collaborative filtering. This paper proposes the use of temporal queries to write such applications, as such queries are easy to specify and naturally real-time-ready. We propose a framework called TiMR that enables temporal queries to scale up to massive offline datasets on existing M-R infrastructure. We validate our approach by proposing a new end-to-end solution using temporal queries for BT, where responsiveness to user interest variation has high value. Experiments with StreamInsight and SCOPE/Dryad using real data from our ad platform validate the scalability and high performance of TiMR and its optimizations, and the benefit of our BT approach in effective keyword elimination, lower memory usage and learning time, and up to several factors better click-through-rate lift.

REFERENCES

[1] D. Abadi et al. The design of the Borealis stream processing engine. In *CIDR*, 2005.

[2] M. Ali et al. Microsoft CEP Server and Online Behavioral Targeting. In *VLDB*, 2009.

[3] B. Babcock et al. Models and issues in data stream systems. In *PODS*, 2002.

[4] M. Balazinska et al. Fault-tolerance in the borealis distributed stream processing system. In *SIGMOD*, 2005.

[5] R. Barga et al. Consistent streaming through time: A vision for event stream processing. In *CIDR*, 2007.

[6] R. Chaiken et al. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2), 2008.

[7] Y. Chen, D. Pavlov, and J. Canny. Large-scale behavioral targeting. In *KDD*, 2009.

[8] T. Condie et al. Mapreduce online. In *NSDI*, 2010.

[9] A. Das et al. Google news personalization: Scalable online collaborative filtering. In *WWW*, 2007.

[10] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[11] D. A. Freedman. *Statistical Models: Theory and Practice*. Cambridge University Press, 2005.

[12] V. Ganti, A. Konig, and X. Li. Precomputing search features for fast and accurate query classification. In *WSDM*, 2010.

[13] A. Gates et al. Building a high-level dataflow system on top of map-reduce: The Pig experience. *PVLDB*, 2(2), 2009.

[14] S. Ghemawat et al. The Google file system. In *SOSP*, 2003.

[15] G. Graefe. The Cascades framework for query optimization. *Data Engineering Bulletin*, 18, 1995.

[16] Hadoop. `http://hadoop.apache.org/`.

[17] M. Hammad et al. Nile: A query processing engine for data streams. In *ICDE*, 2004.

[18] B. He et al. Comet: Batched stream processing for data intensive distributed computing. In *SoCC*, 2010.

[19] J. Hu, H. Zeng, H. Li, C. Niu, and Z. Chen. Demographic prediction based on user's browsing behavior. In *WWW*, 2007.

[20] M. Isard et al. Dryad: distributed data-parallel programs from sequential building blocks. In *SIGOPS/EuroSys*, 2007.

[21] C. Jensen and R. Snodgrass. Temporal specialization. In *ICDE*, 1992.

[22] J. Kramer and B. Seeger. A temporal foundation for continuous queries over data streams. In *COMAD*, 2005.

[23] B. Li et al. A platform for scalable one-pass analytics using mapreduce. In *SIGMOD*, 2011.

[24] C. Loboz et al. Datagarage: Warehousing massive performance data on commodity servers. In *VLDB*, 2010.

[25] D. Logothetis et al. Stateful bulk processing for incremental analytics. In *SoCC*, 2010.

[26] L. Neumeyer et al. S4: Distributed stream computing platform. In *KDCloud*, 2010.

[27] Open Directory Project. `http://dmoz.org`.

[28] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. In *IPM*, 1988.

[29] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *PODS*, 2004.

[30] The LINQ Project. `http://tinyurl.com/42egdn`.

[31] P. Tucker et al. Exploiting punctuation semantics in continuous data streams. *IEEE TKDE*, 2003.

[32] C. van Rijsbergen et al. New models in probabilistic information retrieval. *British Library R&D Report*, 1980.

[33] S. Viglas and J. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD*, 2002.

[34] J. Yan et al. How Much Can Behavioral Targeting Help Online Advertising? In *WWW*, 2009.

[35] Y. Yu et al. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[36] J. Zhou, P. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *ICDE*, 2010.