Overview of Turn Data Management Platform for Digital Advertising

Hazem Elmeleegy, Yinan Li, Yan Qi, Peter Wilmot, Mingxi Wu, Santanu Kolay, Ali Dasdan
Turn Inc.
first.last@turn.com

Songting Chen[*]
Facebook Inc
stchen@fb.com

ABSTRACT

This paper gives an overview of Turn Data Management Platform (DMP). We explain the purpose of this type of platforms, and show how it is positioned in the current digital advertising ecosystem. We also provide a detailed description of the key components in Turn DMP. These components cover the functions of (1) data ingestion and integration, (2) data warehousing and analytics, and (3) real-time data activation. For all components, we discuss the main technical and research challenges, as well as the alternative design choices. One of the main goals of this paper is to highlight the central role that data management is playing in shaping this fast growing multi-billion dollars industry.

1. INTRODUCTION

Over the last decade, a number of radical changes have reshaped the worlds of digital advertising, marketing, and media. The first is an innovation called programmatic buying, which is the process of executing media buys in an automated fashion through digital platforms such as real time bidding exchanges (RTB exchanges) and demand-side platforms (DSPs). This method replaces the traditional use of manual processes and negotiations to purchase digital media. Instead, an ad impression is made available through an auction in an RTB exchange in real time. Upon requests from RTB exchanges, DSPs then choose to respond with bids and proposed ads on behalf of their advertisers for this impression. The entire end-to-end buying process between RTB exchanges and DSPs typically takes less than 250 ms including the network time, leaving less than 50 ms for DSPs to run their runtime pipelines. It is well understood that in order to make such dynamic buying decisions optimal, all

[*]This work was done while the author was at Turn Inc.

data, including user data, advertiser data, contextual data, plays a central role.

The second major shift is the prolific use of mobile devices, social networks, and video sites. As a result, marketers have gained powerful tools to reach customers through multiple channels such as mobile, social, video, display, email, and search. There are numerous platforms dedicated to single channel optimization. For example, video channel platforms aim to maximize the user engagements with video ads, while social ad platforms aim to increase the number of fans and likes of a given product. Regardless of channel, data driven approaches have been proven to be very effective to lift the campaign performance there.

With all the advance of these technologies, one major challenge to the marketers today is that the marketing strategy becomes a lot more complicated than ever before. While a lot of work has been done to optimize each individual channel, how different channels interact with each other is little understood. This is however very important as customers often interact with multiple touch points through multiple channels. One main obstacle is that while there are abundant data to leverage, they are all in different platforms and in different forms. As a result, it is a non-trivial task to create global dashboard by extracting aggregated reporting data from different platforms. Performing even finer grain analytics across channels is virtually impossible, which is the key to campaign effectiveness, accurate attributions, and high return-on-investment (ROI) for the different channels.

Recently, data management platforms (DMPs) have been emerging as the solution to address the above challenge. It is a central hub to seamlessly (and rapidly) collect, integrate, manage and activate large volumes of data. In summary, a DMP needs to provide the following functionalities:

1. Data integration: a DMP should be able to cleanse and integrate data from multiple platforms or channels with heterogeneous schemas. More importantly, the integration has to happen at the finest granular level by linking the same audience or users across different platforms. This is the only way that a deeper and more insightful audience analytics can be obtained across all campaign activities.

2. Analytics: a DMP should provide full cross channel reporting and analytics capabilities. Examples include

aggregation, user behavior correlation analysis, multitouch attribution, analytical modeling, etc. Furthermore, DMP should be delivered through cloud-based software-as-a-service (SaaS) to end users and provide them the flexibility to plug in their own analytical intelligence.

3. Real-time activation: a DMP should be able to not only get data in, but also send data out in real time. In other words, it needs to make the insights actionable. For example, it should be able to run models and perform scoring tasks in real time by combining online and offline data, and to send the output data to other platforms to optimize the downstream media and enhance the customer experience.

In this paper, we will give an overview of a DMP that Turn, a digital advertising startup, has built recently. We hope this overview may provide some support for the fact that DMPs are fine examples of how to handle big data in batch mode as well as real time, unifying techniques from multiple fields of data science, including databases, data mining, streaming, distributed systems, key-value stores, machine learning, etc.

The rest of this paper is organized as the high-level overview of Turn DMP and its three main components: data integration, analytics, and activation. The paper ends with our conclusions and a summary of future work.

2. OVERVIEW

2.1 Audience and Nested Data Model

A user is the central entity in the schema of any DMP. Users in the digital advertising industry are usually referred to as the *audience* (for ads). From a business perspective, advertisers wish to learn everything about their audience: their demographics, psychographics, online and offline behavior, and how they respond to the different types of ads across several channels. Perhaps even more importantly, they wish to understand how all these variables can be correlated. Such a holistic view of the audience data is key to the deign of highly targeted ad campaigns that would maximize their ROI.

From a technical perspective, there are multiple ways in which this data can be captured inside the DMP. The obvious approach is to use the Relational Data Model, where each entity type, including users, will be modeled as a separate table in the relational schema. The main problem with this design arises from the fact that some tables (containing transactional or behavioral data) can be extremely large. With the Internet-scale, the impressions table, for instance, can quickly reach hundreds of billions of records. In this case, correlating variables spanning multiple such enormous tables can be prohibitively difficult.

Fortunately, this type of analysis is typically meaningful only when performed at the user-level. For example, studying the correlation between the frequency of visiting news sites and the likelihood of clicking on stocks-related ads requires evaluating the two variables *for each user*, joining on the user id, and then computing the correlation. So if all the information related to each individual user was already grouped into one big nested record, or *user profile*, the computation would be a lot easier. This observation makes it

clear how a *Nested Relational Data Model* is a better alternative for the types of analysis and the scales involved in a DMP. Even beyond the digital advertising domain, the nested relational data model has already gained wide adoption in the field of big data analytics (e.g., [17]).

In Turn DMP, a user profile covers all available information for a given *anonymized* user, including demographics, psychographics, campaign, and behavioral data. User profile data is typically collected from various sources. It could be first party data (i.e., historical user data collected by advertisers in their own private customer relationship management (CRM) systems), or third party data (i.e., data provided by third party data partners, typically each specializing in a specific type of data, e.g., credit scores, buying intentions, etc.). In that sense, user profiles are treated as a first class citizen and are the basic units for offline analytics as well as for real time applications.

Based on the functionality, Turn DMP maintains two versions of the user profiles. First, the analytical user profile (AUP) is designed for the purpose of offline analytics and data mining. It is stored in Hadoop File System (HDFS) [13]. Second, the runtime user profile (RUP) is stored in a globally replicated key-value store to enable fast and reliable retrieval in few milliseconds for real time applications.

2.2 Real-Time Online Advertising

Figure 1 shows the main players involved in real-time online advertising (the online advertising ecosystem). Whenever a web page of a web site (owned by a publisher) is about to be viewed by a user (a member of the audience) on the user's browser, an ad request is sent to an RTB exchange (public or private) directly or through intermediaries (called supply-side platforms) to find an ad to display on that page.

RTB exchanges serve as the market between the demand side (the side of advertisers and their helpers such as DSPs and DMPs) and the supply side (the side of publishers and their helpers such as supply-side platforms) [4, 6]. RTB exchanges match ads to ad space as well as determine the price of each match. Since RTB exchanges do not have ads, they pass ad requests to multiple DSPs to ask for an ad and the corresponding bid price. Each ad request contains the numerical id for the user and the URL for the web page.

Advertisers use the web console of DSPs to create their advertising campaigns. An illustrated view of the web console is shown in [3]. Each campaign contains at least the following properties: goal (e.g., optimize for impressions, clicks, or actions), budget (total money to spend), spending schedule (e.g., spend evenly per day), targeting constraints (e.g., reach only females in California who have visited a fashion site more than twice in the last month), and ads. Advertisers also use the web console of DMPs to manage the data for their campaigns and the audience they (want to) reach.

When a DSP gets an ad request, it uses the user id and URL in the request to extract the corresponding profiles from its profile servers. It then scans through each ad in the system, filters out those ads whose constraints are violated, and computes a bid for each remaining ad. Finally, it runs an internal auction (usually the first price auction) and submits back to the RTB exchange the winner ad of the internal auction and its bid.
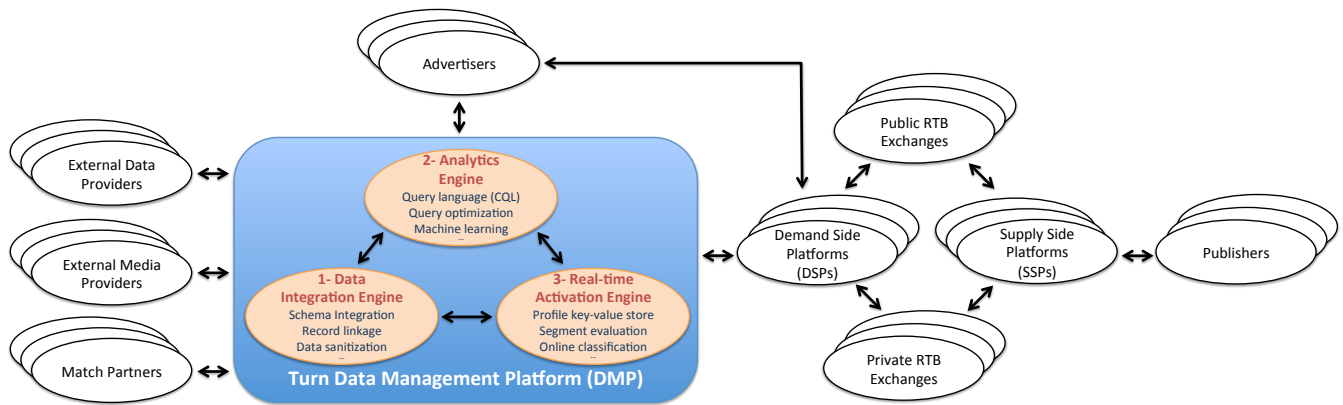
Figure 1: High-Level Design of Turn DMP and Positioning in the Online Advertising Ecosystem
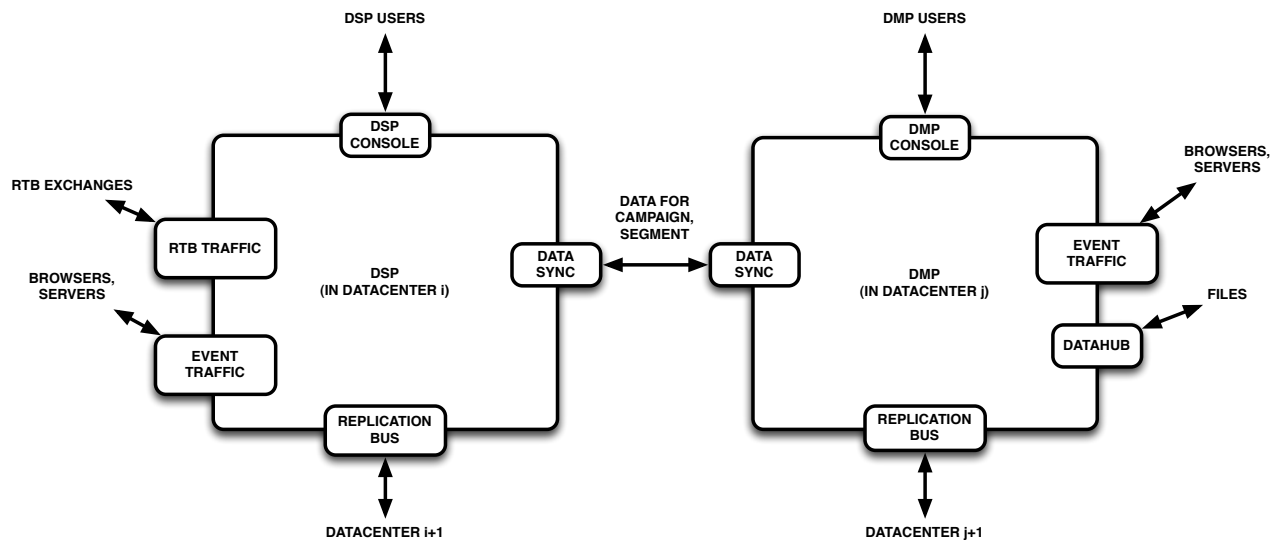

Figure 2: Interactions between a DSP (left) and a DMP (right).

Once the RTB exchange receives responses to its ad request after some set time limit, it runs an auction (usually the second price auction) and sends back to the web page the winner ad. The web page then redirects to an ad server (part of a DSP or independently operated) to get the actual image (called a creative) associated with the winner ad.

The real-time aspects of this entire process is astonishing: The process takes at least three roundtrips through the internet, each one taking up to 80 milliseconds. The time left to each DSP is up to 50 milliseconds, during which the best ad for a given user on a web page (or other online contexts or channels such as video, social, mobile, and search) is selected from millions of ads. The number of ad requests per second, as Turn handles, has recently surpassed a million.

2.3 DMP Architecture

Figure 2 shows the interface of a DSP and a DMP as well as how they interact with each other. Each platform has a web-based console to interact with their respective users.

Inputs to a DSP consist of ad requests from RTB exchanges, events (for impressions, clicks, actions, or data events) from user browsers or external servers, campaign and/or segment data from DMPs, and requests from the console. Outputs of a DSP consist of ads/bids to RTB exchanges, events to user browsers or external servers, and data for the console. Here external servers may belong to the same DSP residing in a different data center or other advertising entities.

Figure 3 details the interface and internals of Turn's DMP. The event traffic (also called front-end or web tier) servers ingest online events fired from browsers or sent from other external servers (e.g., an advertiser's web server indicating that a user has made a purchase). Collected events are sent to the RUP-Store in real time as well as to the AUP-Store in bulk (at regular intervals). The AUP-Store represents the data layer of Cheetah [10], the warehousing and analytics engine in Turn DMP.

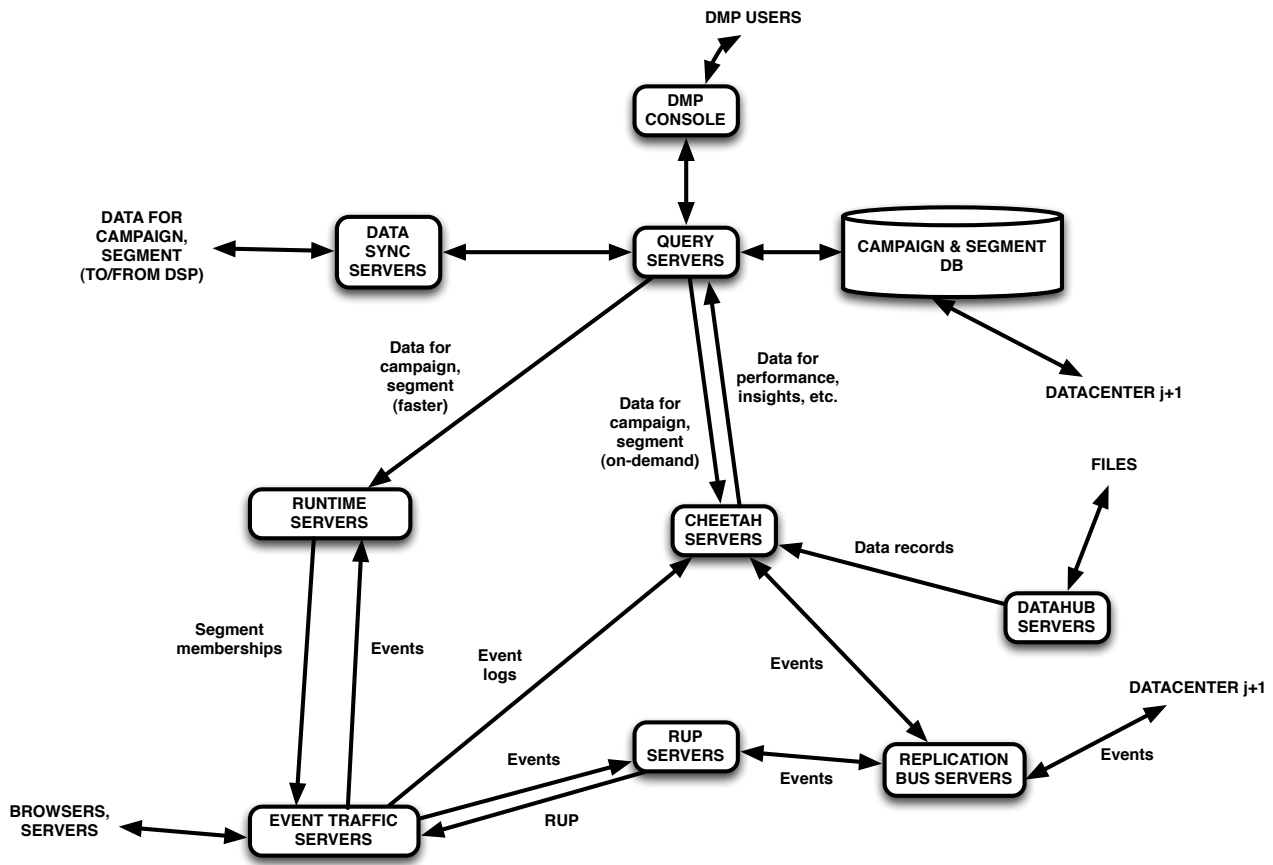Datahub is the component responsible for ingesting offline

Figure 3: Internal Architecture of a DMP.

(file-based) data from first and third party providers, resolving heterogeneity at schema and data levels (e.g., disparate user ids), and performing the necessary sanity checks. Data ingested by Datahub also end up both in the AUP-Store (directly) and in the RUP-Store (through what we refer to as the repletion bus – for global distribution).

A relational database is used to store campaign and segment definitions, campaign performance and insight caches, etc. Query servers are used an an intermediate layer between the relational database and the Cheetah servers on one side and their different clients (e.g., the web console) on the other side. The relational database is replicated to other data centers via the replication technology that comes from the database vendor (e.g., Oracle's advanced data guard).

The external and internal movement of data is managed using dedicated servers such as the data sync servers (to pass the campaign and segment data to DSPs) and the replication bus servers (to pass the collected online and offline data to RUP-Stores).

The *data integration engine* (first high-level component of Turn DMP in Figure 1) involves all the subcomponents related to the collection, ingestion, cleaning, and transformation of both online and offline data, such as event traffic servers and Datahub servers.

The *analytics engine* (second high-level component of Turn DMP in Figure 1), or Cheetah, provides data analysts with a custom, nesting-aware, SQL-like query language called Cheetah Query Language (CQL) in addition to a library

of data mining methods and machine learning models to extract useful insights from the campaign and user raw data.

The *real-time activation engine* (third high-level component of Turn DMP in Figure 1) runs on top of the RUP-Store. It enables different types of computations performed in real-time, such as segment membership evaluation, bid calculation, etc. This is where the extracted insights drive decision making in real-time, or in other words become actionable.

In the following sections, we will explain each of the three components in detail.

3. DATA INTEGRATION

We mentioned earlier that the data stored in a DMP is obtained from multiple sources, including online and offline sources. Integrating all this data incoming at very high rates while meeting the latency and accuracy requirements poses some serious challenges.

Note that in this paper, we use the term "data integration" loosely to cover the tasks of ingesting data from the different input sources, performing the necessary data cleaning, and finally linking and merging the data before it gets stored into the data warehouse with the integrated schema. This is different from how the term is sometimes used in the literature (e.g., [16]) to denote the ability to answer queries posed over a virtual mediated schema by rewriting them on the fly to sets of other queries that can be run on the

underlying physical data sources.

3.1 Integrating Online Data

Online data is collected through front-end servers getting events in real-time about users viewing ads, clicking on ads, making some online actions (e.g., purchase, signup, etc), or visiting a partner's web page and getting a new cookie. Online events can also include RTB requests processed by a partner DSP to bid for placing an ad for a particular user viewing a particular page.

Scalability: The integration of online data poses multiple challenges, starting with the scalability challenge. The front-end servers collecting online data are normally spread across multiple data centers around the globe – to be as close as possible to their client-base. They have to process hundreds of thousands of events every second. They also need to ship their collected data back to the central data warehouse of the DMP in a highly accurate and timely fashion.

These requirements are addressed by using a carefully designed topology, dedicated high-speed links, and applying various WAN optimization techniques [22] to connect the network of data centers. The design can be scaled up as traffic increases by adding more hardware into the data centers or adding new data centers altogether.

Data Quality: Another challenge is related to the quality of online data. For example, click fraud detection techniques (e.g., [18]) are applied to eliminate duplicate (or fraudulent) click events. This step is important to avoid over-charging advertisers for clicks determined not to have any impact on their ROI. It is also an important step to ensure the quality of the data maintained in the DMP, and consequently the quality of the generated analytics from this data.

Event Reconciliation: A third challenge has to do with reconciling different-yet-correlated events, which normally occur at different points in time. The most common example is reconciling a user action along with the click event that led to such an action, and in turn with the impression that marks the beginning of the whole chain of events. Establishing these links is essential to give the credit to the publishers where the ads were viewed and clicked, leading to users ultimately taking action.

In general, this type of data linkage is achieved through tracking ids passed within the HTTP requests and responses. An exciting version of the problem is what is known as *multi-touch attribution* (MTA), whose goal is to link (or to attribute) a user's action to multiple click and impression events that took place prior to the action, potentially on multiple publishers' websites. This is in contrast to giving the full credit to the last click/impression alone. MTA will be described in more detail in Section 4.5.

3.2 Integrating Offline Data

Offline data refers to any information that third party providers may have about users, and would like to make available for advertisers to better target their audience. Part of offline data can also be first-party data that is supplied by advertisers for their own customers; i.e., coming out of their own private CRM systems.

Scalability: Turn DMP is connected to tens of data providers

providing hundreds of types of information to characterize users from many different angles, leading to billions of incoming data records everyday. In addition, all these numbers are growing at a very high rate. It follows that having a scalable infrastructure that can ingest and process these vast amounts of data is crucial.

To address this challenge, Turn DMP enables multiple external and internal FTP connections so that data providers can upload their data sets on a daily basis. Once uploaded, multiple concurrent jobs are launched to copy the data files to HDFS on the Hadoop cluster. Then another set of concurrent map-reduce jobs are started to link the incoming data records to the existing user profiles in the data warehouse, as will be described next. By exploiting the fact that the ingestion and linkage tasks are highly parallelizable, this design can scale well with the data size by making use of more hardware, and hence more parallelism.

Schema Heterogeneity: Clearly, the biggest challenge in integrating offline data is that it can potentially be about virtually anything! In technical terms, this means it can be drawn from any arbitrary domain, have any arbitrary schema, and use any arbitrary vocabulary. Data diversity can span anything from user demographics; to buying, viewing, browsing, and sporting habits; to financial information; to political inclination; to interests in all sorts of subjects; to intensions about, say, buying a new pair of shoes, car, house, etc – to name just a few.

With the increasing number of data providers, and their supplied data sets, a naive solution would attempt to manually create a global schema for the DMP's data warehouse incorporating the individual schemas of all incoming data sets, and then incrementally update the global schema whenever a new data provider/data set is added. This solution will simply not scale.

Instead, Turn DMP requires partner data providers to ensure that their supplied data sets follow a generic schema that can be seamlessly integrated into the data warehouse. The key insight is that all incoming data is meant to *categorize* users, so by definition it is *categorical* data. Based on this observation, data providers are required to build a *taxonomy of categories* for each data set. The main attributes for each node in a taxonomy are: *category_id*, *category_name*, *parent_category_id*, and *taxonomy_id*. As a result, all data sets will be then in the form of lists of user-category assignments. This scheme greatly simplifies the ingestion process of external data without the need for adding any special extensions to the data warehouse schema. At ingestion time, data sanity checks are performed to filter out any data records not conforming with their associated taxonomies, thus shielding Turn DMP from dirty offline data.

It is worth mentioning, that while some data providers may be providing the same type of information about users, Turn DMP intentionally leaves their data sets independent. This approach enables advertisers to choose the data sets of their favorite providers for use in building their ad campaigns. That said, we are also exploring the option of building new clearly-marked data sets that are the outcome of merging multiple data sets from different providers. These derived data sets would potentially have a higher user coverage compared to any of the individual input data sets. Of course achieving this goal involves tackling some hard problems mosty related to ontology matching [20].

Record Linkage: Another challenging problem with the integration of offline data is related to record linkage. Data providers typically use their own user ids in their supplied data records, and hence these records need to be linked to their corresponding user profiles inside the data warehouse. Since the incoming data records usually do not contain any uniquely identifying information beyond the provider's user ids, the problem of record linkage is then resolved using mapping tables.

Fortunately, a number of companies in the digital advertising ecosystem, called *match partners* specialize in building and maintaining such mapping tables across the different data providers and DMPs. Match partners can obtain Personally Identifiable Information (PII) (e.g., email, mailing address, phone number) about users by partnering with publishers having large registration databases. This way they can create records containing their own cookie ids along with the corresponding user's PII. The next step is to link those records to the user records maintained by both data providers and DMPs.

Linking with DMPs is usually done via an online process called *cookie synching.* In this process, whenever the user's browser sends a request to the DMP (e.g., during ad serving), another request is created in the background and sent to the match partner. This second request will have the DMP's user cookie id as a parameter. The match partner will then be able to retrieve its own cookie id from the user's browser (or create a new one if it didn't exist), thereby establishing the mapping to the DMP for that user. As a final step, the match partner can create a third request back to the DMP with its own cookie id as a parameter for the DMP to also establish the mapping with the match partner. All these requests are processed in the background before the user's originally requested web page is returned.

Linking with data providers can be different, especially if they don't have their own cookies (e.g., businesses with large offline databases in the retail or financial sectors, etc). Unlike DMPs, these data providers have the users PII. Thus, linkage is achieved by matching the PII available to both match partners and data providers. Privacy-preserving techniques can be used (e.g., [7]) to ensure that only the minimal information needed for linkage is shared and nothing else.

Problems arise when the mapping tables provided by multiple match partners are in conflict. Reasons may include the inaccuracy or outdatedness of the PII used by some match partners. They may also include the fact that some match partners use PII at the household level rather than the individual level; e.g., mailing address or home phone. Turn DMP can be configured to use one of multiple resolution strategies for this type of conflicts such as: earliest-match, latest-match, majority-voting, and weighted-majority-voting.

Note that in some cases, offline data is keyed off information beyond just the user id. For example, it can be based on the user ip, or it can be location-specific data (e.g., weather data). In these cases, data ingestion is performed directly without the help of external match partners, since the join key information is already available to the DMP.

4. ANALYTICS

As mentioned in Section 2.3, the Analytics engine essentially refers to Cheetah, which is a high-performance data

Figure 4: Part of the Nested Schema Layout and Sample Data for the AUP-Store

warehousing software developed by Turn. It resides at the heart of Turn DMP. An earlier version of Cheetah was covered in [10]. It was also mentioned that the data store that Cheetah manages is denoted by the AUP-Store. Therefore, we might be using the terms "Cheetah" and "AUP-Store" interchangeably in the following discussion.

In this section, we overview some of the key features and optimizations in Cheetah, with an emphasis on the new advances since [10]. For example, the nested relational data model was only briefly mentioned in [10]. In what follows, we will highlight how choosing this model impacts all the different layers involved in Cheetah. In particular, we will describe the schema design, storage format, and data loading aspects of Cheetah. We will also discuss the Cheetah query language and some query optimization methods including the extensive use of materialized views. Finally, we will describe how Cheetah can support custom and built-in machine learning methods for data analysis.

4.1 Schema, Storage, and Loading

Schema: As explained in Section 2.1, we adopt the nested relational data model for the AUP-Store. Figure 4 partially shows the schema and the nested table structure representing the AUP-Store. It can be seen that every AUP, which represents a single record in the AUP-Store, has some basic attributes (e.g., *user_id*, *ip_address*, *browser_type*, etc) and some *nested tables* (e.g., *impressions*, *clicks*, etc). Recursively, each nested table has its own basic attributes and potentially some inner nested tables too. For example, in Figure 4, the *impressions* nested table has an inner nested table, *data_provider_attributions*, which keeps track of all the data providers whose data sets were used during targeting, ultimately leading to the impression event. The targeting and provider attribution processes will be explained in more detail in Section 5. The actual nested schema for the AUP-Store has tens of inner nested tables, and hundreds of attributes.

Storage: The storage layer in Cheetah is currently being switched from the standard row format to the record columnar storage format, similar to [15]. The columnar format (e.g., [12, 15, 17, 21]) is known to bring a lot of benefits, such as higher compression ratios and the ability to skip disk scanning for entire columns once deemed irrele-

vant to the query being processed. With the nested model, the implementation of the columnar format gets more challenging compared to the relational model, particularly because in the former case, not all columns are at the same level in the schema hierarchy.

To overcome this issue in our design, the storage of each column is divided into two parts: a *header* followed by a *payload*. The payload simply contains the list of values in the column (with some compression technique applied as configured by the user). The header contains information on how these values are distributed across each record in the higher schema levels. More precisely, for each higher level in the schema, there is a *sub-header* with as many entries as there are records at that level. Each entry points to the location of the first value in its corresponding record, and also stores the number of values contained in that record.

This design is best illustrated by an example. Consider there are two users ($U1$, $U2$), where $U1$ has one impression ($I11$) and $U2$ has two impressions ($I21$, $I22$). Consider further that $I11$, $I21$, and $I22$ have two records each in their nested table *data_provider_attributions* (or *dpa* for short) with providers: ($P111$, $P112$), ($P211$, $P212$), and ($P221$, $P222$) respectively. In this case, a simplified version (e.g., compression is ignored for clarity) for the storage of the column *aup_store.impressions.dpa.provider_id* can be represented as follows. The payload will be composed of the following list of fields: (6, $P111$, $P112$, $P211$, $P212$, $P221$, $P222$). The header will be of the form: (2, [2, $loc(P111)$, $loc(P211)$], [3, $loc(P111)$, $loc(P211)$, $loc(P221)$]), where [. . .] denotes a sub-header for a certain schema level (*aup_store* and *aup_store.impressions* in this case). In general, each list with multiple items is preceded with the number of items in the list.

Note that the information in the header is sufficient to reconstruct the nested records, mainly by aligning the pointers in each higher level with the pointers in the lower levels to build a hierarchy for the column values.

Since the AUP-Store is stored in HDFS, this large nested table is split into multiple *shards*, where each shard is a separate file containing a subset of the user profiles. These profiles are stored using a PAX-like [9] format, where individual columns are stored sequentially following a file header that has pointers to the start location of each column. Each individual column is stored using the format described above. Similar to Trevni [5], each file occupies a single HDFS block, whose size is set to a large value of 1GB or more. This guarantees the co-location of columns and, at the same time, columns will span a large number of user profiles leading to bigger compression ratios and I/O benefits.

Loading: As discussed in Section 3, Turn DMP receives billions of data records every day both from online and offline sources, representing different types of entities. The data integration engine performs the necessary sanity checks, establishes the required links between certain entities (e.g., impressions, clicks, and actions), and ensures that each incoming record can be associated to a specific Turn user id. The output of the data integration engine is a set of sanitized and properly-linked records for each entity type.

At the end of each day, such records are loaded into the AUP-Store in a two-step process. In the first step, a *delta*

AUP-Store is created for that day. The delta AUP-Store has exactly the same nested structure as the *current* AUP-Store. In particular, all the data records belonging to a given user are used to populate the different nested tables in an initially empty AUP for that user. Similar to the current AUP-Store, the delta AUP-Store is also sharded. It is crucial for the correctness of the second step, as will be shown next, that the sharding strategy used to build the delta AUP-Store is identical to the one used for the current AUP-Store. The default sharding strategy in Cheetah is a uniform distribution, where for n shards, user i is stored in shard ($i\%n$).

The second step is responsible for merging the two stores, delta and current, to create the *new current* AUP-Store. For this purpose, a *merge-join* is used. Since each shard in both input stores cover the same range of users, and since we also guarantee during the creation process that user profiles in each shard are sorted by user ids, the merging process becomes straightforward – each pair of corresponding shards are merged in a single linear scan. Merging two user profiles involves copying all data records from the two input profiles into the proper nested tables of the newly created profile. The entire process is implemented using map-reduce jobs, exploiting the fact that it is inherently parallelizable.

4.2 Cheetah Query Language

The Cheetah Query Language (CQL) was first described in [10]. CQL is an SQL-like query language, where one of its key goals is to be as simple as possible for analysts (especially in the digital advertising world) to quickly grasp and use. In this section, we focus on CQL's extensions to query nested relational data, which was not covered in [10]. Even though CQL can query nested data, its output is still relational and not nested. The key features can be demonstrated through some sample queries.

Consider the example mentioned in Section 2.1 about measuring the correlation between the frequency of visiting news sites and the click-through-rate (CTR) for ads related to stock trading. This analysis can be performed using the following nested CQL query (say for 2012's data).

```
Q1:   SELECT (subquery_1a / 365) news_visits_per_day,
      (subquery_1b / subquery_1c) stock_trading_ads_ctr,
      count(*)
      FROM aup_store
      DATES [2012_01_01, 2012_12_31]
```
where,
```
    subquery_1a =   SELECT count(*)
                    FROM aup_store.impressions
                    WHERE url_type = 'news'

    subquery_1b =   SELECT count(*)
                    FROM aup_store.clicks
                    WHERE ad_type = 'stock-trading'

    subquery_1c =   SELECT count(*)
                    FROM aup_store.impressions
                    WHERE ad_type = 'stock-trading'
```

Q1 builds a two-dimensional histogram with one dimension representing the average daily frequency at which a

user visited news sites in 2012, and the other dimension representing the user's CTR for ads related to stock trading. The histogram counts the number of users falling into each bucket defined by one of the possible combinations for the two dimensions. The correlation between the two variables can then easily be computed from Q1's answer.

Note that each subquery in Q1 operates on one of the nested tables in the user profile. In general, a subquery in CQL can operate on arbitrarily deep nested tables (e.g, *aup_store.impressions.dpa*). Also, the *scope* of subqueries in CQL is implicitly limited to individual user profiles. This is the commonly desired semantics, as can be seen in the example of Q1. Furthermore, if we relaxed this constraint by allowing the correlation of nested tables *across* profiles rather than only *within* profiles, the performance can become extremely slow.

In terms of CQL's succinctness, some simplifications were already described in [10] such as the omission of the GROUP BY clause (where all queries implicitly group by the non-aggregate columns in the SELECT clause), and the use of the DATES clause to easily specify the date range of interest.

Along the same lines, CQL provides a simple option to express subqueries using what we refer to as *list functions*. For example, subquery_1a can be expressed as "*list_count(aup_store.impressions, url_type, 'news')*". Other list functions include *list_sum()*, *list_min()*, *list_max()*, and *list_has()*. Using *list_has()*, whose return type is boolean, corresponds to using the *exists* keyword with subqueries in SQL. Since many of the CQL queries that run in Turn DMP require several subqueries, users were generally appreciative of the simplification offered by list functions.

The arguments for a list function start with the input nested table followed by a number of filters. In the case of *independent* subqueries (e.g., subqueries 1a, 1b, and 1c), each filter is in the form of an attribute name followed by a single constant value or a list of of alternative values. In the case of *dependent* subqueries, at least one filter is in the form of a pair of attributes: the first being from the input table of the subquery, and the second being from the outer query.

To show an example for queries with list functions capturing dependent subqueries, consider Q2 below. This query shows the distribution of the number of unique US users broken down by advertiser, and further, by the frequency of viewing ads from each such advertiser in 2012.

```
Q2:    SELECT advertiser_id,
           list_count(aup_store.impressions,
           advertiser_id, outer.advertiser_id) frequency,
           count(distinct user_id)
           FROM aup_store.impressions outer
           DATES [2012_01_01, 2012_12_31]
           WHERE country = 'USA'
```

Clearly, list functions can only express certain classes of subqueries, and not any general subquery. For instance, filtering is limited to conjunctions of disjunctions. However, our experience with Turn DMP shows that they have been sufficient to fulfill the vast majority of business requirements in the context of digital advertising analytics. It is also worth mentioning that while CQL does not currently support explicit *joins*, much of the needed join functionality (also in the context of digital advertising analytics) can be achieved through the use of dependent subqueries (or list functions).

4.3 Query Processing and Optimization

Query Processing: In Cheetah, query processing is performed using the map-reduce programming model. Each query is executed using a single Hadoop job. The details of query planning and execution for relational data were described in [10]. In a nutshell, mappers are responsible for the selection and projection operations in addition to partial aggregation over their input data splits. Reducers, on the other hand, are responsible for final aggregation and potentially final selection based on the HAVING clause. Moreover, if required, reducers perform local sorting of output records, and then the sorted sublists output by each reducer are merged at a central location before the final answer is returned.

The same principles apply when processing nested queries over nested data. The two main differences are the introduction of: (a) the *unnest* operator, and (b) the *list function* operators.

- *The unnest operator:* Since the FROM clause of a CQL query can refer to nested tables that are arbitrarily deep in the schema hierarchy, and at the same time query results should be relational, it was necessary to introduce the unnest operator during query processing. This operator takes a nested record as input and generates multiple output records from some nested table inside that input record. It also expands each output record with attribute values inherited from the upper-level enclosing record. After unnesting, the expanded records then flow into other operators in the query plan for full processing.

- *The list function operators:* Processing a given list function is somewhat similar to processing the outer query itself. The usual steps of scanning, selection, projection, and aggregation are performed for the specified input nested table. However, all these steps are fully performed in memory given that the scope of each list function is limited to the user profile, which is already loaded in memory by the time a list function is processed. For some list functions, such as *list_has()*, scanning may stop early – as soon as it is determined to return true. Also, independent list functions are computed once for each user profile and then cached. Dependent list functions, on the other hand, which take attribute values from the outer query as parameters, potentially need to be re-computed for each record processed in the outer query.

Query Optimization: In addition to using the nested data model and columnar storage to improve performance, Cheetah also employs a number of optimization techniques to speed up CQL query processing. To name a few,

- *Lazy unnesting:* A naive approach for unnesting is to always fully unnest input records down to the level of the queried table in the schema hierarchy, and then apply selection and other operations. A better approach is to interleave unnesting with selection. Before

unnesting to a deeper level, we first check all selection predicates that are applicable at the current level. If a predicate fails for some upper-level record, then we can safely filter out its lower-level nested records without having to unnest and check each of them individually. We call this approach *lazy unnesting*.

- *Pre-computation of dependent list functions:* Queries with dependent list functions are usually among the most expensive (and hence optimization-worthy) queries. We considered three possible implementations for processing a dependent list function: (a) *nested loop*, where for each record in the outer query, the list function is computed from scratch, (b) *cached nested loop*, which is similar to the previous approach except that for each parameter combination, the list function is computed once and then cached (in a hashtable) – to be looked-up by subsequent records in the outer query with the same parameter combination, and (c) *hash-join*, where the list function is pre-computed for all possible parameter combinations in a single pass – also resulting in a hashtable to be looked-up by records in the outer query. Note that if the outer query table has p records with p_u unique parameter combinations and the inner query table has q records with q_u unique parameter combinations, then the complexity of the above three approaches can be shown to be as follows: (a) $O(pq)$, (b) $O(p_u q + p)$, and (c) $O(p+q)$. This clearly shows the superiority of the pre-computation/hash-join approach, which is used in Cheetah.

- *Predicate re-ordering:* For queries (and subqueries) with multiple selection predicates, Cheetah re-orders those predicates based on their estimated selectivity and processing complexity. This way, highly selective and low-cost predicates are checked first, which lowers the probability of having to check the other predicates at even higher costs.

- *Multi-query execution:* Cheetah allows multiple queries to be submitted simultaneously and executed in a batch mode, given that they operate on the same input data. This data is scanned only once for all those queries. Also the deserialization cost, as well as the setup cost for mappers and reducers, is incurred once. In addition, if some of these queries share the same list functions, then they share their computation too.

- *Materialized views:* Cheetah makes extensive use of materialized views to improve query performance. The following subsection is dedicated to explaining the different types of materialized views used in Cheetah.

Figure 5 gives an example that illustrates the query processing and optimization aspects in Cheetah. It shows the query plan for executing Q2, and how it is distributed across multiple mappers and reducers. The lazy unnesting optimization is applied since only user profiles with the correct country (USA) are unnested further. Otherwise, they are discarded. Also the *list_count()* function is pre-computed for all possible *advertiser_id*'s and the result is hash-joined with the filtered impression records from the outer query. Finally the grouping and aggregation steps are performed first locally in each mapper and then globally in the reducers.
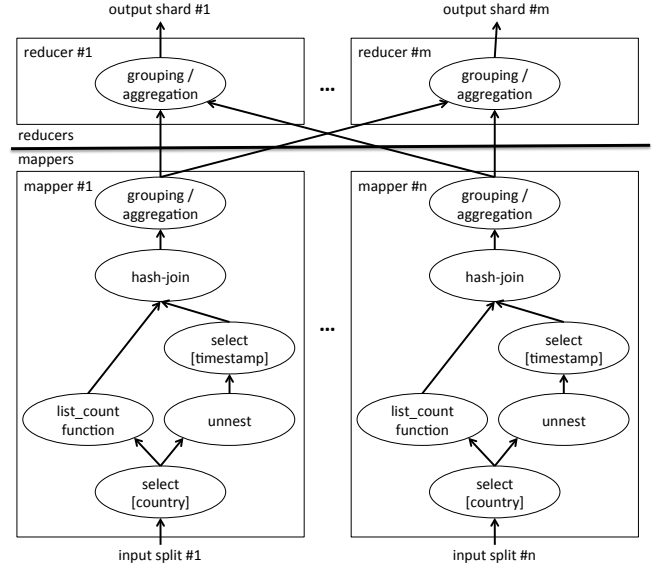

Figure 5: Distributed Query Plan for Q2 with n Mappers and m Reducers

4.4 Materialized Views

The goal of building materialized views is to save some pre-processing effort during query execution. Before executing a query, the query planner in Cheetah, considers multiple execution plans for that query. In particular, it decides whether the query needs to run on the full AUP-Store, or if one of the existing materialized views can be used to answer the query. This step is usually called *view matching*. If the query matches more than one view, then the one expected to minimize the cost is used. At this point, a second step called *query re-writing* is performed, where the query is re-written to be expressed in terms of the materialized view instead of the full AUP-Store. This is generally a hard problem [14], but in our special context, it is usually feasible, as will be shown shortly.

At a high level, two different types of materialized views are used in Cheetah: *aggregation-based* and *partitioning-based*. The partitioning-based materialized views are further divided into *vertical partitioning*, *horizontal partitioning*, and *combined partitioning*.

Aggregation-Based Materialized Views: These materialized views use the relational data model. They typically summarize the performance of advertising campaigns. Each view is defined by a collection of dimensions and metrics. So for example, a materialized view can capture for each advertising campaign by each advertiser in each region on each day, and so on, the key campaign performance metrics such as the total number of impressions, clicks, and actions.

A query is considered to match one of these views if its grouping and filtering attributes are a subset of the view dimensions, and its aggregate columns match the metrics in the view. Aggregation-based views normally get higher priority in answering matching queries compared to partitioning-based views. This is because in the former case, a good amount of the query processing has already

been done. In the latter case, the benefits mainly come from fully processing the query over a smaller input.

Vertical Partitioning Materialized Views: Unlike aggregation-based views, these views have a nested relational data model similar to the full AUP-Store. Specifically, they have a pruned version of AUP-Store's hierarchical schema. This leads to smaller-sized versions of the AUP-Store that can be scanned and processed faster.

We considered two types of vertical partitioning: *coarse-grained* and *fine-grained*. By coarse-grained, we mean that vertical partitioning occurs only at the boundaries of the higher-level nested tables in the AUP-Store schema. For example, tables like *aup_store.impressions* or *aup_store.clicks* are either completely included in a given vertical partition or completely excluded.

The fine-grained version, on the other-hand, enables the selection of individual columns (leaves in the schema hierarchy) from any nested table to include in a given vertical partition. To identify the best fine-grained vertical partitions to build with respect to the query workload, we extended a workload-aware vertical partitioning algorithm [8] to handle nested data. The details of the algorithm are beyond the scope of this paper.

A query is considered to match a certain vertical partition if it includes all the columns referred to by that query. The smallest vertical partition matching a query is used as the input for query processing. While coarse-grained vertical partitioning is easier to implement, fine-grained vertical partitioning is more flexible and hence can achieve bigger gains in query processing – given the same budget of disk space.

Horizontal Partitioning Materialized Views: Similar to vertical partitions, horizontal partitions also have a nested relational data model. However, unlike vertical partitions, they have the exact same hierarchical schema as the AUP-Store. The key difference is that the data records themselves are partitioned based on some partitioning attribute(s).

We also considered two types of horizontal partitioning: *deep* and *shallow*, depending on whether the partitioning attributes are at deep levels in the schema hierarchy or only at the top level. For example, deep horizontal partitioning can be applied on the AUP-Store based on *advertiser_id*. This implies creating a separate AUP-Store for each advertiser. Of course, nested tables with no *advertiser_id* will have to be replicated in all partitions. This way, queries interested in some advertiser-specific information can only focus on the partition for that advertiser. Note that in this type of horizontal partitioning, the same user may exist in multiple partitions – as their profile will be split across all those partitions.

By shallow horizontal partitioning, we mean that each user profile will exist in a single partition only. One possible way to achieve this type of partitioning is to cluster users based on their activity life spans. Users with similar earliest and latest activity times (top-level attributes) will fall into the same partition together. Thus, queries interested in certain time intervals can skip partitions whose earliest activity time is beyond the interval of interest or whose latest activity time is before the interval of interest.

Combined Partitioning Materialized Views: We note that the strategies for vertical partitioning, deep horizontal partitioning, and shallow horizontal partitioning are all orthogonal, and hence can be combined. In other words, it is possible to have a single partition having a schema with only a subset of the columns in the full AUP-Store, focusing only on a specific advertiser, and including users whose activity life spans are very similar – all at the same time. This combined partitioning scheme can generally achieve the best results in terms of decreasing the amount of data needed to process for answering queries, and consequently in terms of the overall query performance.

4.5 Advanced Analytics

CQL allows for SQL-based aggregations and correlations between different audience events. Sometimes, marketers look for more advanced analytics, such as modeling and machine learning. One good example is multi-touch attribution (MTA) [19].

MTA is a billing model that defines how advertisers distribute credit (e.g., customer purchase) to their campaigns in different media channels (video, display, mobile, etc.). For example, suppose a user sees a car ad on her web browser. Later, she sees a TV commercial about the same car again, which makes her more interested. Finally, after she sees this ad again on her mobile phone, she takes the action and registers for a test drive. Marketers know that all these media channels contribute to the final conversion of an audience. However, the current common practice is last-touch attribution (LTA), where the last impression, the one on the mobile phone, gets all the credit. A better and more fair advertising ecosystem is expected to distribute the credit to all the channels that contributed to her final action. This is the so-called multi-touch attribution problem. In Turn DMP, different MTA models are incorporated as user defined functions (UDFs) into CQL. This way CQL users have the freedom to feed the MTA algorithm with any arbitrary input data.

As a final remark, CQL as well as the data mining UDFs are exposed to external clients as a data service in the cloud so that they can perform ad-hoc analysis and obtain very unique insights on their own. Some examples are logistic regression, k-means clustering, and association rule mining.

5. REAL-TIME ACTIVATION

The main purpose of the real-time activation engine is to use the data available about users (also about web pages and other types of data) as well as the insights built on top of this data to drive bidding decisions in real-time. This section describes some design aspects of the RUP-Store, which makes all these types of data available in real-time. It also describes the different types of computations needed to support decision making at bidding time, and how these needs are addressed.

5.1 Runtime User Profile Store

Similar to AUPs, RUPs also have a nested data model to capture the different types of information about users. However, the amount of information stored inside each RUP is usually much smaller than its corresponding AUP - to enable faster retrieval and processing. Generally, the most recent information gets higher priority to be kept in the RUP.

The RUP-Store is a high-performance key-value store that was developed by Turn, with keys being user ids and values being RUPs. It is designed to provide low-latency read/write access, typically within a few milliseconds to support a peak 1,000,000 queries per second across multiple, geographically distributed data centers.

The design of the RUP-Store is inspired by Dynamo [11] and Voldemort [2]. In particular, we have built a software layer on top of Berkeley DB (BDB) that uses consistent hashing to achieve sharding, replication, consistency, and fault tolerance. It also employs flash drives as physical storage – since hard disks are not fast enough for our real-time applications. The RUP-Store is replicated locally within each data center as well as globally across data centers to achieve high availability and local low-latency access.

Specifically, to guarantee real-time synchronization of RUPs across data centers, we have built an infrastructure called the replication bus (See Figure 3) that incrementally replicates user events across data centers and distributes them to profile stores to keep the different replicas of the RUP-Store synchronized. The replication bus is horizontally scalable and highly optimized to be able to process tens of billions of events daily between data centers with an average end-to-end SLA of within a few seconds.

Moreover, unlike most general-purpose key-value stores, the RUP-Store enables the incremental update of existing RUPs whenever new user events arrive through both online and offline data ingestion. Upon event integration, specific business logics may be applied locally in the RUP servers, e.g., imposing a cap on the number of events stored for each user, or doing pre-aggregation on raw events. This approach is much more efficient than applying business logics on the client side, which requires costly exchanges of full RUPs back and forth between the clients and the RUP servers.

Finally, it is worth mentioning that the Turn-developed key-value store can be used to maintain other types of profiles beyond just RUPs. For example, it can be used to maintain profiles for web pages, weather data, stock-market data, etc.

5.2 Real-time Processing

Recall from Section 2.2 that from the DSP's perspective, given a user id and a page URL, the two key decisions to make are: which ad to display on this page, and how much to bid for it. Making those two decisions can potentially trigger a series of requests to the activation component of the DMP to be answered in real-time.

In particular, since each currently running ad campaign is defined by a user segment, then it is desired to know which segments the user in the bid request belongs to. This requires a lookup in the RUP-Store to get the user profile, followed by multiple segment evaluations against that profile. User segments are typically complex logical expressions defined on the user attributes, and are provided by the advertisers. The user attributes used in segment definitions can either capture the user's online behavior or be part of the data ingested offline for that user.

Some common examples of online behavior attributes are the *frequency* and *recency* attributes. Frequency represents the number of, say, ad impressions the user made in, say, the past week, while recency represents how long ago the user made her last ad impression for instance. Note that computing this type of attributes may require non-trivial

processing.

When segment definitions include attributes based on third party ingested data, then the additional processing of *data provider attribution* is also needed. The goal of this attribution step is to identify how much credit each data provider should get in case the bid is won and the ad impression is made. For example if a segment is defined to be the ANDing of the four attributes: A, B, C, and D, where A, B, and C are provided by provider $P1$, while D is provided by provider $P2$; and if it was determined that a given user belonged to that segment and she was served an ad accordingly, then $P1$ would get 75% of the credit and $P2$ would get the remaining 25%. Making this computation is important to fairly share the revenue with the data providers.

Another example use of real time computation on user profile is evaluating a user against some machine-learned models. These models can be specified by the users of the DMP in some proprietary format or by using industry standard model specification language such as Predictive Model Markup Language (PMML) [1]. An example model can be something that either predicts a car buyer based on latest online activity or a person likely to apply for a credit card. Having such knowledge in real time is immensely valuable to Turn clients as they can use these predicted signals to bias the campaigns or take other actions in real time.

In addition to ad selection, which is focused on finding the ad campaigns matching the user in the bid request, deciding on the bid price may also require processing the user profile. A machine learning model, for instance, may predict that a given user is very valuable and hence worth a high bid price, or vice versa.

All these examples highlight the intensive real-time processing requirements, along with the high and ever-growing rate of requests that the real-time activation side of a DMP has to deal with. They also highlight how challenging building such systems can be.

6. CONCLUSIONS AND FUTURE WORK

Digital advertising has now reached a state where the pipeline between publishers on the supply side and advertisers on the demand site includes many technology partners to help publishers and advertisers deal with real-time optimal decisioning at huge scale. Among these technology partners, data management platforms occupy a prominent role as the hub where all data relevant to reaching the audience over different channels is integrated, analyzed, and shared. In this paper, we have given a high-level overview of Turn DMP as an example demand side platform. We hope that it can serve as a starting point for further exploration.

We predict that due to the efficiencies gained through real-time decisioning and the scales involved with more online usage, the future of advertising will be more real-time. This implies more data and components in real time. We are planning to take our DMP in this direction.

7. REFERENCES

[1] Data management group. Predictive modeling markup language. http://www.dmg.org/, Feb 2013.
[2] Project voldemort, http://www.project-voldemort.com/voldemort/.
[3] The New York Times. The high-speed trade in online audiences. http://goo.gl/IxJZT, Nov 2012.

[4] The New York Times. Your online attention, bought in an instant. http://goo.gl/KZQH0, Nov 2012.

[5] Trevni: A column file format, http://avro.apache.org/docs/current/trevni/spec.html.

[6] TY. Mansour, S. Muthukrishnan, and N. Nisan. Doubleclick ad exchange auction. CoRR, abs/1204.0535, 2012.

[7] R. Agrawal, A. V. Evfimievski, and R. Srikant. Information sharing across private databases. In *SIGMOD*, pages 86–97, 2003.

[8] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD*, pages 359–370, 2004.

[9] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *Proceedings of VLDB*, pages 169–180, 2001.

[10] S. Chen. Cheetah: A high performance, custom data warehouse on top of mapreduce. *PVLDB*, 3(2):1459–1468, 2010.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[12] A. Floratou, J. M. Patel, E. J. Shekita, and S. Tata. Column-oriented storage techniques for mapreduce. *PVLDB*, 4(7):419–429, 2011.

[13] Hadoop. *Open Source Implementation of MapReduce. http://hadoop.apache.org/.*

[14] A. Y. Halevy. Answering Queries using Views: A Survey. pages 270–294, 2001.

[15] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *ICDE*, pages 1199–1208, 2011.

[16] M. Lenzerini. Data integration: A theoretical perspective. In *PODS*, pages 233–246, 2002.

[17] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

[18] A. Metwally, D. Agrawal, and A. El Abbadi. Duplicate detection in click streams. In *WWW*, pages 12–21, 2005.

[19] X. Shao and L. Li. Data-driven multi-touch attribution models. In *KDD*, pages 258–264, 2011.

[20] P. Shvaiko and J. Euzenat. Ontology matching: State of the art and future challenges. *IEEE Trans. Knowl. Data Eng.*, 25(1):158–176, 2013.

[21] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-store: A column-oriented dbms. In *VLDB*, pages 553–564, 2005.

[22] Y. Zhang, N. Ansari, M. Wu, and H. Yu. On wide area network optimization. *IEEE Communications Surveys and Tutorials*, 14(4):1090–1113, 2012.