Kodiak: Leveraging Materialized Views For Very Low-Latency Analytics Over High-Dimensional Web-Scale Data

Shaosu Liu
Turn, Inc
shaosu.liu@turn.com

Bin Song
Turn, Inc
bin.song@turn.com

Sriharsha Gangam
Turn, Inc
sriharsha.gangam@turn.com

Lawrence Lo
Turn, Inc
larry.lo@turn.com

Khaled Elmeleegy
Turn, Inc
khaled.elmeleegy@turn.com

ABSTRACT

Turn's online advertising campaigns produce petabytes of data. This data is composed of trillions of events, e.g. impressions, clicks, etc., spanning multiple years. In addition to a timestamp, each event includes hundreds of fields describing the user's attributes, campaign's attributes, attributes of where the ad was served, etc.

Advertisers need advanced analytics to monitor their running campaigns' performance, as well as to optimize future campaigns. This involves slicing and dicing the data over tens of dimensions over arbitrary time ranges. Many of these queries need to power the web portal to provide reports and dashboards. For an interactive response time, they have to have tens of milliseconds latency. At Turn's scale of operations, no existing system was able to deliver this performance in a cost effective manner.

Kodiak, a distributed analytical data platform for web-scale high-dimensional data, was built to serve this need. It relies on pre-computations to materialize thousands of views to serve these advanced queries. These views are partitioned and replicated across Kodiak's storage nodes for scalability and reliability. They are system maintained as new events arrive. At query time, the system auto-selects the most suitable view to serve each query.

Kodiak has been used in production for over a year. It hosts 2490 views for over three petabytes of raw data serving over 200K queries daily. It has median and 99% query latencies of 8 ms and 252 ms respectively. Our experiments show that its query latency is 3 orders of magnitude faster than leading big data platforms on head-to-head comparisons using Turn's query workload. Moreover, Kodiak uses 4 orders of magnitude less resources to run the same workload.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 9, No. 13
Copyright 2016 VLDB Endowment 2150-8097/16/09.

1. INTRODUCTION

Online advertising campaigns seek best performance, which can be defined as: given a fixed budget, maximize some goal. The goal could be as simple as showing maximum number of ads to users having certain criteria. It can also be more sophisticated like maximizing amount of sales for a certain product or subscription to some service. A key control to maximize these goals is setting the targeting criteria. This is a non-trivial task as each user often has hundreds or thousands of attributes. Crafting the right criteria allows for reaching the exact audience of interest and hence maximizing the impact for a given budget.

Optimizing campaigns' performance is often an iterative process. It involves monitoring running campaigns to make sure they are progressing properly. If performance is unsatisfactory, targeting criteria can be tuned to improve performance. Moreover, historical data of previous campaigns can be analyzed to craft the targeting criteria of future campaigns.

Campaigns produce tremendous amounts of data. This data is comprised of events produced by users. Example events are ad views (impressions), clicks, actions, etc. At *Turn*, we process many billions of events per day. Doing advanced analytics over this sheer volume of data, while expecting interactive response time to be served via a web portal was challenging. No preexisting system was able to do meet the required level of service in a cost-effective manager.

Hadoop based query engines, like Pig [19], Hive [27], and Cheetah [7], can scale to handle very large datasets relying on its cost effective Hadoop Distributed File System (HDFS). However, they have the fundamental latency limitation of Hadoop's MapRecuce [13], where query latency is at best in the order of minutes.

Newer non-MapReduce-based engines like Impala [5], Shark [31], SparkSQL [4], and Presto [28] provide lower latencies as they rely on more efficient execution engines and better utilize memory. However, they do not deliver sub-second query latency [29]. This is fundamentally due to how they are architectured. A distributed query is constructed of multiple tasks launched at different nodes. These tasks work together to execute the query. Task creation and management introduces an inherent latency and overhead. Moreover, all of these systems lack out-of-the-box creation and maintenance of views as well as view selection at query time. Further-

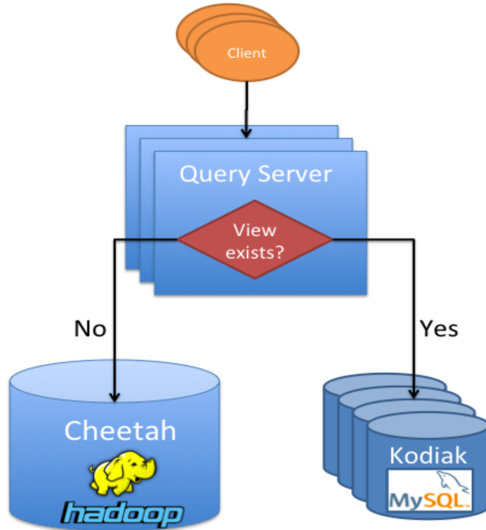


Figure 1: Query routing in the Kodiak analytical plaform.

more, like other Hadoop-based systems, they lack out-of-the-box indexing support requiring full data scans, even though most queries have high selectivity, which unnecessarily inflates query latency.

Multiple traditional commercial Massively Parallel Processing (MPP) warehousing systems, e.g. TeraData and Netezza, have the facility of building and maintaining materialized views. Once available, they also select the best view to serve a query for minimum response time. Moreover, they have indexing support to expedite high selectivity queries. However, these systems typically run on specialized hardware for maximum performance. Consequently, they are often much more expensive than newer systems running on commodity hardware. Moreover, it is questionable that they can scale to maintain thousands of views for multi-petabyte dataset with hundreds of dimensions to achieve sub-second response time. Even if this was possible, it would be prohibitively expensive to most.

To support analytics slicing and dicing this high dimensional data over arbitrary time ranges in an interactive yet cost effective way, we took a hybrid approach. First, we relied heavily on pre-computation, aggressively creating materialized views for different data cubes. Given the high dimensionality of the data, thousands of materialized views were needed. Hence, we needed a scalable platform to manage all these views. Consequently, we built a system, Kodiak, whose sole responsibility is building, maintaining, and querying views. It is horizontally scalable as it can support more views by adding more nodes to its cluster. Moreover, individual views are indexed for fast querying. Kodiak can be viewed as a cache sitting in front of the primary raw dataset (stored on HDFS). As shown in Figure 1, queries that can not be served off Kodiak's views are routed to our ad-hoc general-purpose query engine, Cheetah, to be served off the raw data at HDFS.

Kodiak is distributed analytical database, where time is treated as a first-class citizen. It is horizontally scalable, where views can be time-partitioned and replicated for scalability and reliability. The system maintains the views, up-

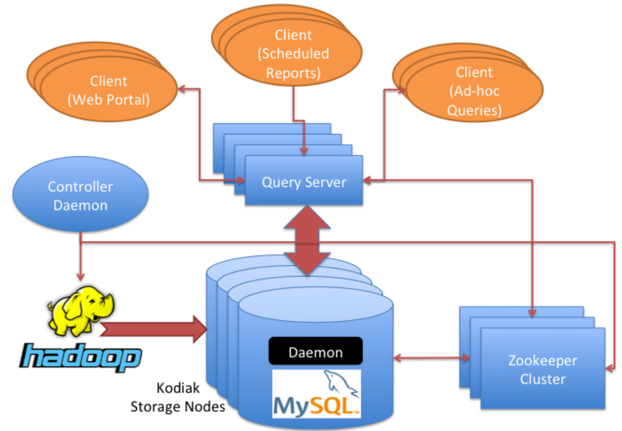


Figure 2: Overview of the Kodiak ecosystem with all the key components involved in its operation.

dating the underlying tables as new events arrive. It is also responsible for view selection, choosing the best view serving a query to minimize response time. View selection is done at runtime. This is useful as it does not have to rely on a fixed list of views available. For example, if a view becomes unavailable, e.g. due to some intermittent failure or delay of the asynchronous process updating it, the system chooses the next best view available.

Kodiak is most useful for serving canned queries for reports from the web portal requiring very fast response time. It is also used to accelerate ad-hoc queries by trying to serve queries that it has views for.

The rest of the paper is organized as follows. Section 2 presents the design of Kodiak. Section 3 presents its implementation. Section 4 evaluates Kodiak's performance as well characterizing our query workload. Related work is presented at Section 5. Section 6 concludes the paper. Zookeeper

2. DESIGN

Kodiak is a highly scalable distributed read-only database for very large-scale high-dimensional analytical queries. It is not meant to be a general purpose analytics platform. Instead, it is purpose built for online advertising data for Turn's use case, which is very large scale (PBs), high dimensional, temporal dataset. It is used to serve canned queries powering Turn's dashboards and reports. In addition, it powers exploratory ad-hoc queries. It is mainly used to store, maintain, and query different summaries (views), rolled up across different dimensions or time granularities.

Raw data is stored on the HDFS and a periodic batch Extract-Transform-Load (ETL) pipeline is used to construct or update the views at the Kodiak cluster. The ETL pipeline transforms raw events with complex schema into a simple schema that is easy to query by analytical queries. This simple schema is denormalized by materializing all needed joins. In addition, the ETL creates and maintains materialized rolled up views (data cubes)[1] of the data for fast query

[1]In the rest of the paper *cubes* and *views* are used interchangeably.

answering. This ETL pipeline is written as a workflow of MapReduce and Pig jobs and is controlled by an external driver. It is configurable in a way that it accepts user-defined cube definitions specifying the required dimensions and measures. Definitions are written manually based on a careful study of the query workload to achieve maximum speedup given the available resources. Aided by the graph at Figure 3, we arrived at the best set of cubes that maximize our query coverage given our space budget. The ETL's optimizer leverages shared execution and shared scans to eliminate redundant work, when computing and updating the views representing overlapping data cubes. Views are maintained and updated incrementally as new raw data arrives. For better performance, updates are not done in real time to allow raw data to accumulate in meaningful batches to maximize sharing of execution, while maintaining the data freshness business requirements. In other words, we adopt the deferred view maintenance approach for its superior performance [8]. Song *et al.* [26] describes the detail ETL pipeline and data population of Turn, while this paper focuses on view storage and query performance.

Kodiak daemons pick up these updates from HDFS to update the corresponding view tables. Depending on their size, views of fact tables may be time (range) partitioned, where shards are distributed among Kodiak nodes.

At query time, the system selects the best view to query for best performance. In the following sections, we will cover system architecture and operations of different components in more details.

2.1 Kodiak's Architecture

As shown in Figure 2, a Kodiak cluster is composed of multiple storage nodes. Each storage node runs a relational database instance (MySQL). In addition, it runs a Kodiak daemon responsible for maintaining this instance. Daemons coordinate and receive instructions via a Zookeeper cluster [15]. When the ETL pipeline produces new updates to the views, it assigns them to different Kodiak nodes. It then notifies the corresponding daemons via Zookeeper. Daemons load their updates concurrently off HDFS to update the relevant views on their local databases. Newly created/updated views are marked ready for querying once their data is loaded.

Views are stored as database tables on the underlying database nodes. Based on the configuration, each view can be sharded and replicated, with a configurable replication factor. If sharded, views are partitioned based on events' timestamps. Views and shards are assigned to different Kodiak nodes by the ETL pipeline. Aided by information about sizes and usage of views, the pipeline tries to distribute the views/shards in a way that balances the load across Kodiak nodes with respect to storage and query load.

Kodiak also maintains metadata about stored views. For each view, the metadata tracks its shards as well as the physical tables backing them on the corresponding database. If the view is replicated, it also tracks different replicas. If the view has multiple versions, the metadata tracks these versions too. Metadata is 3-way replicated across 3 storage nodes for fault tolerance. Updates to the metadata, when views are updated, are asynchronously replicated to other replicas relaxing consistency for better performance. We use master-master replication for that. This has the benefit of simplifying our design and implementation as well as increasing the performance of metadata updates. Any live replica can be used as the master to write updates to. Consequently, if the current master dies, another replica is chosen with no further action required. Master-master replication is suitable for this use case as there is one writer to the database, so there are no conflicts. This single writer is the ETL, when it updates the mappings in the metadata after each iteration.

As a consequence of asynchronous replication, a client may query an older version of a view if it uses metadata from a replica that is behind. However, this may only happen for very brief periods of time as replicas should catch up promptly. This only slightly affects data freshness. In any case, Kodiak does not promise realtime data, so this is inconsequential.

For availability, if one storage node hosting a metadata replica goes down, another storage node is selected to replace it. Metadata is replicated to it asynchronously. A cluster master daemon is responsible for monitoring replicas and in the event of failure, it is responsible for selecting an alternate node to host the data of the failed node. The master also acts as the driver of the ETL pipeline. It is responsible for launching the pipeline periodically.

2.2 Selecting Views To Materialize

Views are built to cache the computation of constructing them, i.e. build once and query many. Specifically, many of the queries, if executed against the raw data, would involve very expensive operations like joining very large fact tables together (Tables of raw events like clicks or impressions), or doing rollups over very large tables. Caching the computation improves overall system throughout as it eliminates redundant work. In addition, it substantially reduces query latency, allowing responses to be available virtually instantaneously.

The ETL pipeline is responsible for creating and maintaining these views. For its inputs, it operates on the schema of fact tables of raw events and their corresponding dimension tables. It produces a simplified denormalized schema, with all required expensive operations precomputed. This simplified schema is what is exposed for querying.

2.2.1 Formal Problem Definition

Choosing which views or cubes to materialize can be modeled as the following optimization problem. Find the best set of views to materialize to minimize the weighted average query latency for our query workload, while respecting our space limitations as well as respecting the Service Level Agreement (SLA) for SLA-bound queries.

Formally, it can be stated as follows. Given an overall set of queries Q_{all}, with a subset Q_{sla} representing queries having tight SLA (e.g. queries servicing a web portal with low latency requirement), and a set of materialized views V:

$$\begin{aligned} \underset{q_i}{\text{minimize:}} \quad & \sum_i w_i \delta_i \quad \forall q_i \in Q_{all} \\ \text{subject to:} \quad & \\ & latency(q_j) \leq L \quad \forall q_j \in Q_{sla} \\ & \sum_k size(v_k) \leq S \quad \forall v_k \in V \end{aligned} \tag{1}$$

w_i is the weight of query i. For example, canned queries serving the web portal could be given higher priorities, which

translates to higher weights. Also, queries serving a more important web page, e.g. front page, can be given even higher weight than other canned queries. δ_i is the average query latency for query i. Q_{all} is the set of all queries served by the platform. This can be approximated using the list of queries seen at our query logs. In addition, any incoming new canned queries can be added as well. Conversely, Q_{sla} is the set of all SLA-bound queries. Hence it should be a subset of Q_{all}. L is the maximum allowed latency for a SLA-bound query. V is the set of views chosen to be materialized and $size(v_k)$ is the size of view k.

2.2.2 Examples Of Materialized Operations

The two most notable operations that are materialized are joins and rollups. The ETL materializes all expensive instances of these operations.

Other than joins with dimension tables, there are two key join operations between huge fact tables that are prohibitively expensive to do at query time.

Click Deduplication: This operation eliminates duplicate clicks by the same user to the same advertisement. Duplicates are often due to software bots. In any case, advertisers care for one click per advertisement per user. To cost bound this very expensive operation, we time bound the look-back window searching for duplicates to a fixed amount, e.g. thirty days. Duplicates are eliminated using an auxiliary table maintaining users' histories keyed by the user id. This technique allows for the incremental elimination of duplicates as when new clicks arrive, they are joined with the auxiliary table.

Action Reconciliation: *Actions* represent users doing an event desirable to the advertiser, e.g. buying an item. In this case, the advertiser sends an event to Turn called *beacon*. Beacons' fact table is joined with impressions' fact table to construct actions. An action record is generated if and only if a new beacon record matches a preceding unmatched impression. Similarly, a fixed look-back window is used to cost bound this join, e.g. 30 days. Also similarly, the join is done with the help of an auxiliary user history table to allow the operation to be performed incrementally as new events arrive.

2.2.3 Choosing Data Cubes To Materialize

Most of the queries involve doing rollups on a set of dimensions. Given the sheer volume of raw data, evaluating these rollups from scratch at query time is very expensive and often prohibitively expensive. To solve this, different data cubes are materialized. Given the high dimensionality of the data (many hundreds of dimensions), the number of possible cubes is tremendous. Kodiak's scalable architecture allows us to manage and store a very large number of cubes. Nevertheless, our platform is optimized to be cost effective. In other words, for a given space budget, it tries to maximize its query coverage. Aided by business knowledge, query logs, and statistics in Figure 3 we arrive at the best set of dimensions to materialize rollups for for maximum query coverage.

The most naive approach is to have one large cube comprising all these dimensions in question. This has two shortcomings though. First, given this cube is very large, some rollup queries may require a significant additional amount of aggregation at query time if only a smaller set of dimensions is needed. This can violate the web portal's SLA. Consequently, for web-portal canned queries, given these queries are known a-priori, other coarser-grained cubes are added to materialize rollups further. This consumes more space but in practice, the finest-grained cube/table dominates the space requirements, especially that individual canned queries usually only involve a handful of dimensions. Second, having one large cube comprised of all the dimensions of question assumes that any subset of dimensions can be used together. In practice though, we have learned that some of these dimensions are mutually exclusive. This created the opportunity to partition this large cube into multiple smaller ones. Even though these smaller cubes had overlapping sets of dimensions, some very high cardinality dimensions were mutually exclusive, which hugely reduced the size of the cross product of these dimensions resulting in much less space requirements to store these smaller cubes. For example, the two dimensions geo-location and the domain name at which the advertisement was served are not used simultaneously at queries in practice. Each of these dimensions have tremendous cardinality. So, separating them into separate cubes reduces the overall space requirements significantly. Specifically, for our workload, having one large cube requires 25TB of space. Partitioning it into three smaller cubes as explained above, reduced the total space requirements to 9.5TB – a 62% reduction.

2.3 View Maintenance

Materialized views are typically summaries of data rolled up across different dimensions or date ranges. As new raw events arrive, e.g. new impressions or clicks, they are logged on HDFS, then the relevant views need to be updated. Similarly, when events pass their retention periods, they need to be purged off their corresponding views.

The controller daemon drives the ETL pipeline by periodically launching it to maintain these views by updating their summaries. The pipeline includes multiple jobs. Each job is launched with a minimum acceptable frequency to update its corresponding summaries. For example, for a rollup job updating a daily summary, it needs to run once a day. Whereas for a monthly summary, the job only needs to run once a month. Given the batch nature of these jobs, views' data is not guaranteed to be fresh. This is a pragmatic tradeoff we made, trading freshness for performance, which was acceptable from a business perspective. It is worth noting that this is a common practice in data warehousing as more freshness usually involves higher costs. Depending on the business needs, after some point one starts receiving diminishing returns from more freshness. Finally, the ETL pipeline has an optimizer that tries to maximize the shared execution, as well as the scans of input data for all the maintenance jobs.

Jobs often have to update summaries of older time periods (opposite to just newly unprocessed time period since last iteration). This is because some events arrive many days late because of many of the complexities of the inter-workings of different ad-serving platforms spanning multiple companies. An example of events arriving late is *Actions*. Actions can have different definitions depending on the use case they are serving. A possible definition of an action is a shopping cart checkout after an advertisement has been served. To construct an action event for this, a shopping-cart-completion event needs to be matched with its corresponding impression event in a preprocessing phase. The action is given the

timestamp of the impression[2], which can be many days before the actual purchase is made. Consequently, view maintenance includes insertions of new rows as well as updates to existing rows.

Each iteration/run updates its corresponding view incrementally. It produces new updates on HDFS. The controller daemon then assigns updates and new views to different Kodiak nodes. It tries to balance the load as well as guarantee that replicas lie on different nodes. Then, it notifies the Kodiak daemons, via Zookeeper, that there are updates ready for pickup from HDFS. Zookeeper These views power Turn's web portal, which needs to be always available. Consequently, Kodiak is designed to always keep its views available for querying, even while being updated. Moreover, for consistency, it guarantees that all updates to a view from one ETL iteration are transactional. In other words, it provides snapshot isolation. To accomplish this, each maintenance iteration recreates the updated shards of each view. Specifically, if a shard needs to be updated at the current iteration, a new version is created for it. New (updated) shards are then bulk loaded to Kodiak as new tables at the end of the iteration. Finally, Kodiak's metadata is updated to point to the newly loaded tables as the backing store of the updated view/shard. Only at this point, the updates become visible for querying. Note that we made the design choice to overwrite parts of the view that may have not changed since the last iteration and hence doing redundant work. The cost of this is mitigated by bulk loading opposite to updating individual records in the view that have changed. Also, if a view did not change since the last iteration, it is not updated. Our design gives up some performance for availability.

In more details, updates are loaded into Kodiak as follows. After the ETL produces the updates on HDFS, it updates Kodiak's metadata at Zookeeper signaling that there are new updates ready for pickup. Kodiak daemons, monitoring Zookeeper, get the signal and each bulk loads its corresponding updates, which are new tables constructing the view's new version. After loading is done successfully, daemons update metadata mappings making them point to the new tables holding the new version of the shards/views. Kodiak always maintains two versions of each shard/view. This is useful for Kodiak clients, which cache copies of the mappings of views to tables. Even if a client's cached mapping becomes stale, the client can access the view. It will just hit the older version of the view. Since clients update their mappings periodically, quickly they will get the new mappings.

Finally since Kodiak maintains two versions of each shard or view, after loading daemons check if more than two versions exist. If this is the case, older versions are dropped to free space.

2.4 Query Execution

As shown in Figure 2, queries to Kodiak first arrive at a cluster of query servers. The cluster is composed of identical servers. For load balancing, each client picks one of the servers and submits its query to it. This cluster along with the Kodiak nodes running MySQL instances as well as the Kodiak daemons constitute Kodiak's distributed execution engine. Queries typically aggregate data, grouping it over a set of dimensions and/or time. Filters are also often applied on dimensions, e.g. :

```
Select AdvertiserID, CampaignID,
Count(Impressions), Count(Clicks) from EventsTable
where (AdvertiserID = 123) and (CampaignID = 789)
and (Date >= '02/25/2015')
Group By AdvertiserID, CampaignID
```

In other words, queries only involve dimensions, measures, and filters applied to the dimensions. This is because the pre-constructed simplified schema captures all the needed information from the raw events, making it possible to query it with such simple query constructs. This simplified schema and query structure, largely simplify execution at query time. Further, this simplifies adoption of optimizations, like what is described in Section 2.5, making execution much more efficient.

After the query server selects the best view to serve this query as explained in Section 2.5, aided with information about the locations of shards for the view in question, it time-partitions the query into one or more query fragments. Each fragment is then sent to the storage node hosting the current version of the corresponding view shard, where it is executed locally on the local database server. Result fragments are then returned to the query server, where it does the final aggregation, sending the final result to the client.

The mapping of shards to storage nodes is maintained by all query servers. They periodically load it from Kodiak's metadata tables. Again, a relaxed consistency model is used here. However, this only slightly impacts data freshness. For example, mappings can briefly point to older versions of views/shards. This is inconsequential for this application though.

2.5 Answering Queries Using Views

At a high level, whenever a query server receives a query, it tries to select the best view to answer it. The best view is the one having most of the query precomputed, allowing for the response to be returned fastest. To this end, the query server tries to identify the smallest view that can answer the query as usually this means that most of the query's rollup work is already precomputed. For simplicity, we use heuristics to identify the smallest view or cube. The first heuristic is the number of dimensions the cube has as it is an approximate proxy for the cardinality and hence the cube's size. Query servers maintain a list of available views as well as the columns they cover. When receiving a query, the server extracts the required columns and they are matched against available views, selecting the one with minimum number of dimensions. If multiple views with minimum dimensions can serve the query, heuristics about which dimensions have smaller cardinality are used to select the smallest view. These heuristics are based on the understanding of our schema and workload, e.g. geo-location, and domain name at which the ad was served are known to have the highest cardinalities.

After the view is selected, based on the data range in the query, Kodiak nodes hosting the relevant shards are identified. Then, the query is rewritten for each shard to only query the data available at this shard. After receiving the re-

[2]This is because, like impressions, actions should be confined to the campaign's date range. Purchases happening after the end of the campaign, but resulting from ads belonging to the campaign should still be attributed to the campaign.

sponses from all the shards, the query server compiles them into a single response, sending it back to the client.

2.6 Fault Tolerance And Recovery

Failures in this environment are not uncommon given its complexity. A common root cause is Hadoop failures due to software or hardware problems.

Fault tolerance for the controller daemon is fairly straightforward. For example, after the controller daemon comes up it can detect whether it is recovering from an interrupted/failed iteration or it was gracefully shutdown. This is because for each successfully completed iteration a flag is set (implemented as a file on HDFS). If there are updates from the last iteration on HDFS, but no corresponding completion flag is found, this means the previous iteration failed. In this case, the controller does a roll-forward recovery of this iteration. More specifically, it reruns the ETL pipeline instructing it to only generate the missing updates/views from the failed iteration. Then, it instructs the Kodiak cluster to load these updates and finally writes the completion flag. If the daemon finds the completion flag for the last iteration, it just moves forward to the next iteration as no recovery is required.

Failures of storage nodes are handled by the controller. The controller monitors the liveness of the storage nodes through Zookeeper. If a node is detected to have failed, it redistributes its load on other live storage nodes. This is to prevent under-replication of the shards of the views the failed node owned. It does this by asynchronous replication from another live shard. For subsequent updates from maintenance jobs, it routes them to the newly selected replica via Zookeeper.

Rebalancing happens after the node is detected to have been down for an extended period of time. This is to avoid unnecessary expensive rebalancing if the node's failure is ephemeral. After a storage node comes up, it checks with Zookeeper if it has updates that it is supposed to load. If this is the case, it loads them from HDFS to maintain its data freshness.

2.7 Discussion

Kodiak is custom built for Turn's analytics use case. Aided with intimate knowledge of our workload and constraints, we were able to tailor our design choices to serve these needs best. Even though the system is not general purpose, we believe that many of the lessons learned can be useful to a broader audience.

In the rest of this section, we will discuss our design choices with respect to the overall architecture as well as our choice for the backing store.

2.7.1 Alternative Design

In our design, we chose to have a separate system that is responsible for maintaining the different views. This system is outside our data warehouse, i.e. HDFS, where all our raw data resides. An alternative design could have been to have a unified system responsible for maintaining both the raw data as well as the data views – the same way traditional data warehouses do it. Had we used Kodiak, with mysql nodes as its backing store, as our unified warehouse, we could have leveraged its querying capabilities for view maintenance instead of building something else. However, it would still be challenging doing rollups over arbitrary dimensions for very large datasets as this involves grouping large numbers of records across database nodes. In addition, there are multiple other reasons, mostly efficiency related, that pushed us away from this alternative.

First, raw data has to reside on HDFS as it is accessed by other kinds of jobs, e.g. doing different kinds of machine learning computations like classification or clustering. So, duplicating the raw data across two systems will be wasteful, especially given its very large size. Second, view maintenance is compute intensive as it involves updating thousands of materialized views in our case. Given our relaxed freshness requirement on the views, doing the view maintenance in a batch mode becomes a natural fit. This allows for multi-query optimization and shared execution. Unfortunately, this is not supported by Kodiak's underlying query engine – mysql. Writing our own MapReduce view-maintenance engine allowed us to exploit the overlap across these maintenance queries as they update overlapping data cubes. Moreover, running these maintenance jobs on Hadoop is more efficient as it allows us to multiplex a big shared compute resource (large Hadoop cluster) instead of provisioning Kodiak's storage nodes for these heavy tasks. Furthermore, this performance isolates interactive queries from maintenance ones providing more deterministic response times. Finally, it is cheaper to store raw data on raw flat files as it does not require having and maintaining indices for it, which makes HDFS a natural fit for it.

2.7.2 Alternative Backing Store

Another design alternative was to construct the views on Hadoop, but store them and serve them off a different system than a traditional database – mysql in our case. For example, they could have been stored on a NoSQL platform like HBase [2] [3] or Cassandra [3] [4], where it offers a global out-of-the-box index making queries run much faster. Also, they provide out-of-the-box sharding and replication for fault tolerance and scalability. However these systems do not have a query engine. So to support arbitrary queries over the views, we need to implement our engine. Another alternative is to store the views on HDFS and then use a platform like SparkSQL or Impala to query them. This way we leverage the same storage infrastructure for both raw data as well as the views. The draw back of this approach is that these systems do not offer the low sub-second latency needed by our application. This is mainly due to two reasons: (1) lack of indexing and (2) task maintenance during query execution. Consequently, we chose traditional database nodes as our underlying backing store.

3. IMPLEMENTATION

In our current implementation, storage nodes are running MySQL RDBMs. Asynchronous replication of metadata is done via Tungsten [30], which is an open source replication engine. Daemons are implemented in Java.

In our previous implementation, we used Oracle Clusterware [21] for the storage layer. Clusterware abstracts a cluster as a single machine. Even though Clusterware has multiple servers serving requests, four in our case, they all share

[3] BigTable's [6] open-source implementation.

[4] Dynamo's [11] open-source implementation.

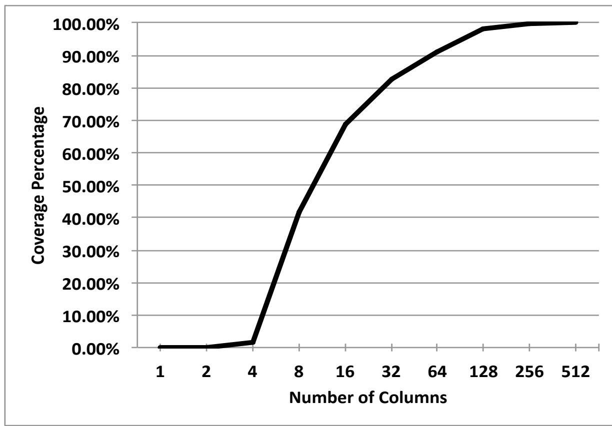


Figure 3: This graph shows how much coverage in our query workload is achieved by subsets of the columns in our dataset. The x-axis has logarithmic scale. It lists all the columns appearing in our query workload sorted descendingly by popularity. The y-axis shows the cumulative query coverage achieved by adding more columns.

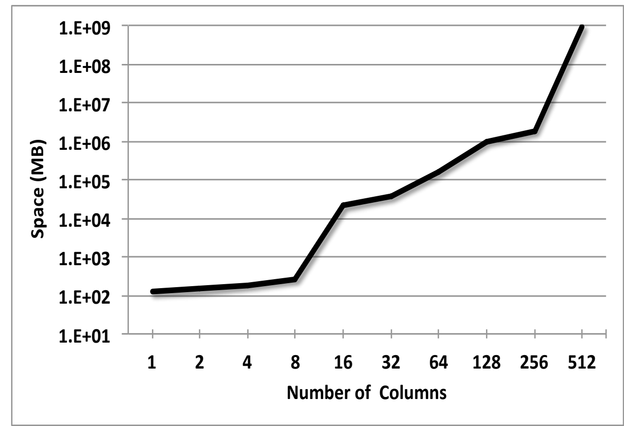


Figure 4: This graph shows the minimum amount of space required to materialize views covering the the different numbers of columns. Both x-axis and y-axis have logarithmic scale. X-axis lists all the columns appearing in our query workload sorted descendingly by popularity. The y-axis shows the minimum space required by views to cover the corresponding columns.

the same disk, which can become a bottleneck. More specifically, updating the views can be time consuming as all updates go to the same disk. This will be covered in details at Section 4. Our newer implementation is significantly superior as the system is fully distributed and read and write operations going to different nodes do not content on shared resources.

4. EVALUATION

This section provides the experimental evaluation of different aspects of the Kodiak system.

First, we give a workload characterization showing the distribution of the popularity of different columns in our data set with respect to the query load. We also quantify the minimum amount of space needed to materialize cubes covering these columns. In these studies, we consider a 24 hour trace of our query workload.

Second, we evaluate the performance of the system with respect to querying as well as view maintenance. Many of the experiments conducted compare the performance of Kodiak to its predecessor described in Section 3.

All measurements are made off the following two setups. The Kodiak cluster has 40 commodity storage nodes. Each server is running MySQL version 5.5.28 instance. They all have Solid State Drives (SSD) for better performance. The previous generation system is run on a 4-node Oracle Cluster version 11g release 11.2. The cluster uses Oracle's Automatic Storage Management (ASM) [20]. This storage layer has 27 disks, where data is stripped and replicated for performance and reliability. The cluster is connected to the ASM layer via a Storage Area Network (SAN).

4.1 Workload Characterization

In this experiment, we present a study of our workload, which we used to optimize our system choosing which views or cubes to maximize our coverage and consequently maximize our performance gains.

Figure 3 shows how much coverage in our query workload is achieved by subsets of the columns in our dataset. In other words, the x-axis lists all the columns appearing in our query workload sorted descendingly by popularity. The y-axis shows the cumulative query coverage achieved by adding more columns. Note that, the x-axis has logarithmic scale. We note that a small fraction of the columns (64 columns) achieves very large query coverage (over 90%).

Moreover, some of these columns are never queried together and hence they form disjoint sets. Consequently, multiple smaller views (with fewer number of columns) can be used instead a larger one to achieve the same coverage. For example, if we have n columns, we can use two views with r and q columns to achieve the same coverage, where both r and q are smaller than n and the columns of both views are subsets of the n columns. The views with smaller number of columns usually have significantly smaller cardinalities resulting in significant reduction in size. If the aggregate size of the two smaller views is smaller than the larger view, then it is a win. In addition, smaller views have better query performance than larger ones as data is preaggregated more. This happens often in our workload, where the aggregate size of the partitions is smaller than the large unpartitioned view. By carefully studying our workload and then exploiting this, we achieve better query coverage and performance using much smaller amount of space.

Figure 4 plots the minimum amount of space required to materialize views covering the different numbers of columns. Similarly, the x-axis lists columns sorted descendingly in the order of their popularity in the query workload. The y-axis shows the minimum space required by views to cover the corresponding columns. Both axes have logarithmic scale. Note that the amounts listed is the minimum amounts of space required to cover the corresponding columns. For better performance, more views are added. So in practice, the amount of space used is usually more. By monitoring and studying our workload, we add more views for queries requiring speedup. These are summary views though, so usually they have smaller cardinalities than the basic views. Hence, they usually use less space.

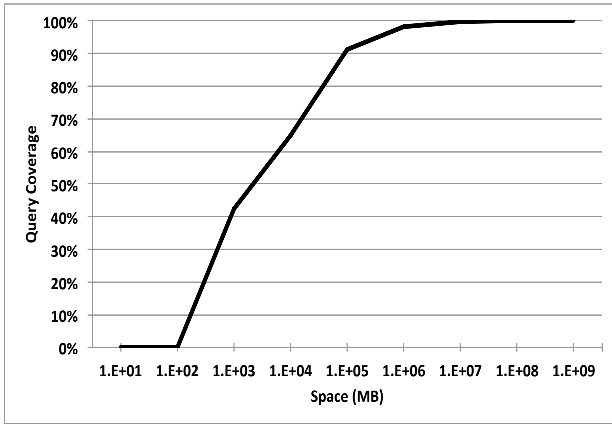


Figure 5: This graph shows the space required to materialize views to achieve different query coverage. The x-axis lists required space. The y-axis shows the query coverage with the different space.

Scenario	# Tables	Space (GB)
A	140	11.74
B	189	45.19
C	70	53.13
D	433	117.90
E	103	126.64
F	438	194.38

Table 1: Details for different view maintenance scenarios

Figure 5 plots the minimum amount of space required to materialize views to achieve maximum query coverage. This figure is produced by combining data from Figure 3 and Figure 4. We note that with a fairly small amount of space (1 TB), we can cover over 98% of our query workload.

Also note that it takes under 1 PB to get 100% query coverage, while we have a raw corpus of data of over 3 PBs. This is because raw data is cleaned and only a fraction of it is exposed to our query system.

4.2 View Maintenance Evaluation

In this section we evaluate the performance of view maintenance/creation. Since our design choice was to trade performance for availability by rewriting entire tables holding modified views/shards, we wanted to evaluate the cost of these updates. To this end, we study the performance of six different production-inspired scenarios of view maintenance/creation. Each scenario has a combination of views created/updated with different sizes. Table 1 shows the detailed information for our evaluation scenarios.

For each scenario, its corresponding tables are loaded into both systems, Kodiak and its predecessor, representing a variant of a view maintenance iteration. Each scenario is run for 3 days on production data against both systems: Kodiak and its predecessor. Average loading times were recorded. Figure 6 shows these results. We see that Kodiak has a significantly better performance than its predecessor with a speedup exceeding 4X in some cases. As expected, with increasing the amount of data loaded, the speedup increases too. This is because of Kodiak's scalability leveraging many underlying database instances.

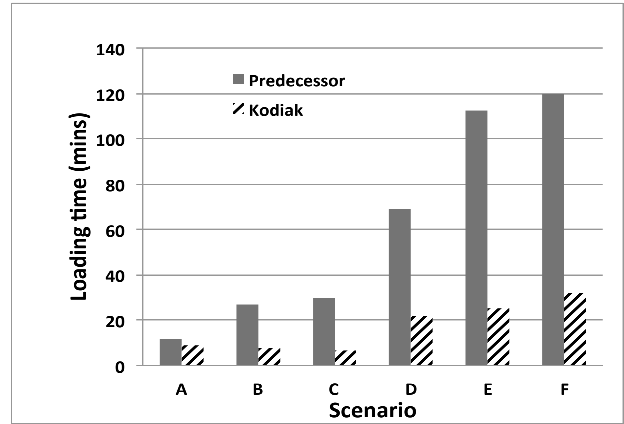


Figure 6: Average loading times for different scenarios for both Kodiak and its predecessor.

Overall Maintenance Cost In Production

In production, one iteration updating all the views produces 8TB of updates data. Data is two way replicated, so the actual size loaded to Kodiak is 16TB. Even though views are updated incrementally, this large volume is generated each iteration due to some events arriving late, resulting in updates to data from older dates. Bulk loading into a distributed cluster mitigates the cost of the update process.

4.3 Query Performance Evaluation

In this section, we study Kodiak's query performance by analyzing the distribution of queries' latency.

4.3.1 Query performance In Production

In production, we serve about 200,000 queries daily. Figure 7 shows the distribution for different query response times measured at our query servers. Note that the x-axis has logarithmic scale. From the figure, we see that our system has median response time of 8 ms and a 99 percentile response time of 252 ms. Note that this graph has capped to query latencies to only 40 minutes. However in our workload, some queries have latencies going over 10 hours. The graph's long tail is capped for presentation purposes. Queries with these very high latencies are a tiny fraction of the overall query population (under 0.1%). These queries are served by the Cheetah system running on top of Hadoop as they do not have views to cover them and hence the very high response time.

4.3.2 Kodiak Vs. Predecessor

In this section, we compare Kodiak's query latency to that of its predecessor described at Section 3. Hence, we exclude queries served by the Cheetah system from this workload. Unlike its predecessor, Kodiak has multiple share-nothing instances. While this distributed architecture offers horizontal scalability, which is critical for this workload, in some cases it comes at a slight cost. For example, Kodiak might compile a query result from multiple result fragments received from different instances. This compilation happens outside the RDBMS, introducing some extra overhead. Figure 8 shows the distributions of query latencies for both Kodiak and its predecessor. In this experiment, we only

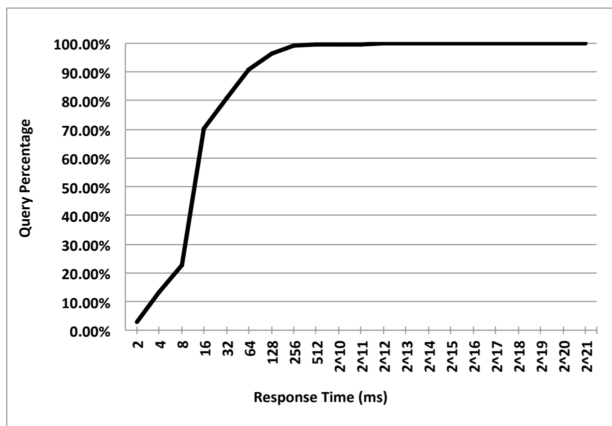


Figure 7: Distribution of query latency in production. This includes queries served by the Kodiak cluster as well the ones served by the Cheetah query engine. Note that x-axis has logarithmic scale. Also, note that the x-axis is capped to 40 minutes. However, the maximum time experienced for a query was over 10 hours. Larger values are capped for presentation purposes. Queries with these very high latencies are a tiny fraction of the overall query population (under 0.1%)

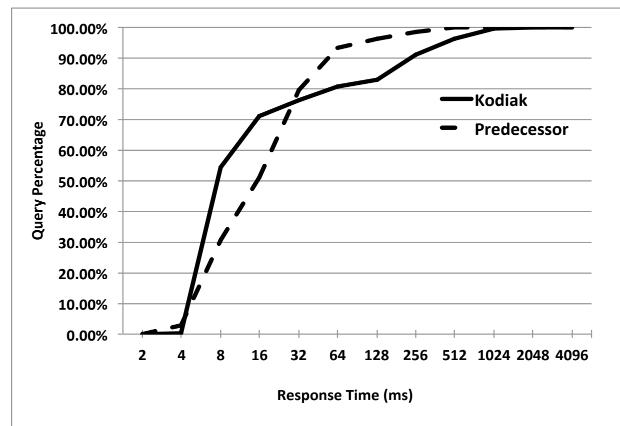


Figure 8: Kodiak performance evaluation Vs. previous generation for the query user cases when Kodiak hits multiple instances.

consider queries, where responses arrive from multiple Kodiak instances to study the effects of partitioning and distributing the data across multiple machines. We note that at the head of the distribution function, Kodiak has lower latencies due to parallelism. However, we also note that Kodiak's query latency distribution function has a heavier tail. This is because the more instances involved in the query, the more likely one of them will get unlucky and stall due to a system issue (e.g. disk or network heavy activity), delaying the entire query response.

4.3.3 Kodiak Vs. Cheetah Vs. Spark

In this section, we compare Kodiak to Cheetah, Turn's adhoc MapReduce-based query engine, to Spark, which represents the next generation, high performance big data query engines. We compare two key metrics: query latency and resource consumption by queries, which is a proxy for system throughput. To this end, we use a family of queries very popular in our workload. For Cheetah, queries are executed on data in raw events form. For Kodiak, queries are executed against materialized data cubes. For Spark, queries are executed against the same data cube used for Kodiak. The only difference is that the cube is materialized on HDFS – Spark's standard data source.

The family of queries used have the form:

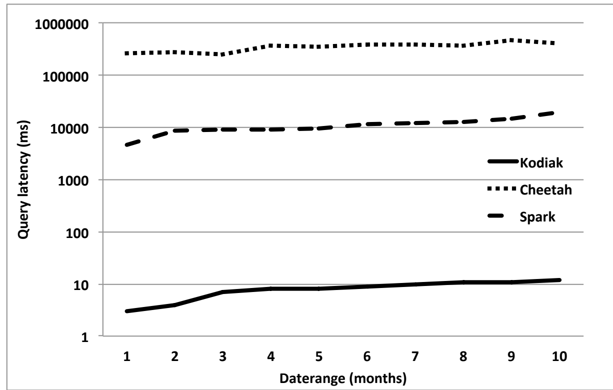
```
Select AdvertiserID, Count(Impressions),
Count(Clicks), Count(Actions) from EventsTable
where (AdvertiserID = 123)
and (Date >= date1) and (Date < date2)
Group By AdvertiserID
```

Cheetah runs on a 640 node Hadoop cluster. Whereas Kodiak runs on the cluster described above. Spark runs on a 10 node test cluster. This cluster could not accommodate all the raw data, so we limited the Spark experiments to ones running on materialized views. The view size used was approximately 1GB, which could fit easily on the cluster – both from storage and processing perspectives.

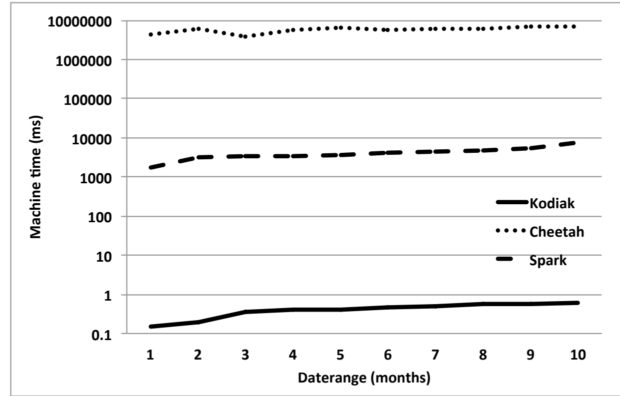
Figure 9 shows the results of running the above query on the three platforms, when varying the date range from one to ten months.

Figure 9a studies the query latencies across the three platforms. We note that Kodiak is three orders of magnitude faster than Spark and more than four orders of magnitude faster Cheetah. This is mostly due to the superior relational database (MySQL) performance with key features like indexing. Conversely, Spark has to scan all the data. Also, its execution relies on distributed tasks, which is costly to launch and coordinate. Cheetah's experiments, unlike those of Spark and Kodiak, run on raw events data opposite to materialized views. Hence, they have to aggregate and filter all events by AdvertiserID at query time. This explains its very high latency. Also, note that all queries rollup data on the time dimension at query time depending on the requested date range. This explains the increased latency for all platforms as the date range increases. In conclusion, neither Spark nor Cheetah or any other Hadoop-based query engine is suitable to serve a workload like Turn's with a tight latency SLA for a web portal. Conversely, Kodiak is able to easily support it.

For multiple reasons, it is hard to have an apples-to-apples comparison of throughput of these different systems. First, each runs on different hardware optimized for its own architecture. Second, it is hard to come up with workload that would not favor one platform over the other. For example, for Kodiak, certain queries hit specific nodes, depending on where the required data is located. To guarantee that the workload evenly spreads across all nodes, it requires careful design of its queries as well as the dataset, which is non trivial. To avoid this problem, we adopt a different approach than directly measuring throughput. Figure 9b studies the resources needed by the three platforms to execute a query. Basically, this can be a proxy for throughput. Given that the three systems are very heterogeneous with different execution models, we tried to normalize their resource usage to a unified metric (Machine.millisecond). For Cheetah, the metric is approximated as follows. Hadoop has



(a) Comparing Kodiak Vs. Cheetah Vs. Spark with respect to query latency time in milliseconds.



(b) Comparing Kodiak Vs. Cheetah Vs. Spark with respect to estimated machine time in milliseconds.

Figure 9: Kodiak Vs Cheetah Vs Spark performance comparison. Note that the y-axis has logarithmic scale.

the concept of slots, where each machine is split into slots, where Hadoop executes its tasks. For each completed job, Hadoop reports the total slot milliseconds it used. Dividing this by the number of slots per machine, we get the total Machine.milliseconds the job took. Similarly, Spark has the concept of virtual CPUs. Dividing the reported vCPU milliseconds by the number of vCPUs per machine, we get the total Machine.milliseconds the job took. For Kodiak, it is a little harder as MySQL does not have the concept of evenly dividing the machine across queries. To approximate this, we used the number of physical CPUs per machine as a proxy for that as most queries are executed sequentially in a single thread. Consequently, we computed this metric by dividing the query latency by the number of physical CPUs per machine multiplied by the number of machines (shards) were used to execute the query. We recognize that our model may have some inaccuracies. However, this was the best way we could come up with to compare resource usage across very different systems. While our results are approximate, we are confident that they are qualitatively accurate, especially given the number of orders of magnitude difference between different platforms.

In Figure 9b, we note that Spark consumes four orders of magnitude more machine ms than Kodiak. Cheetah, consumes seven orders of magnitude more machine ms. Also, we note that resource consumption increases as the date range increases due to the increased work of aggregation over time. So, latency aside, we also conclude it is very costly to execute a workload like Turn's solely on Spark or Cheetah.

5. RELATED WORK

The classical way of doing large scale analytics is using MPP systems. MPP systems like TeraData, Netezza, and Vertica are very established systems in this arena. However, it is questionable if they can support sub-second latencies for complex analytical queries over web-scale data without materialized views. For high-dimensional data, thousands of views could be needed. It is questionable that these classical MPP systems can scale to handle this number of views for web-scale data. Even if this was possible, it is questionable it would be able to scale to handle hundreds of thousands of queries per day in addition to maintaining all these views. Finally, if all of this is possible, it would be prohibitively expensive to many as these systems rely on high-end proprietary hardware.

Another approach is to use NoSQL [17] systems like HBase or Cassandra. These systems solve the scaling issue faced by RDBMS. However, they have the problem that they do not support SQL. Both systems provide a distributed index, but only provide very rudimentary key value operations. Aggregation and group by operations could only be done on client side. Cassandra provides its own query language CQL, which is not very mature and could not fulfill our need. In addition to the lack of SQL support, HBase and Cassandra also present an operational challenge. Since we are loading large amount of data, data compactions are inevitable. During our tests data compaction on both these tools is very resource intensive; it sometimes renders the system unusable when the compaction is underway.

MapReduce based query engines like Pig, Hive, and Cheetah provide rich querying facilities. However, they are fundamentally limited by Hadoop's latency. Newer systems like Impala, Presto, and SparkSQL support low latency SQL as they do not run on top of Hadoop's MapReduce. However, they do not automatically provide sub-second latencies for web-scale data due to lack of native indexing and materialized views support.

Systems like PNUTS [9] and Espresso [22] were built to be horizontally scalable, while relying on traditional RDBMS as their backing stores. This way they harness the power of RDBMS, when the queried data sits in one RDBMS instance, while supporting web-scale. Similarly, F1 [25] was built to be horizontally scalabale and fault tolerance but it was built on top of Spanner [10]. However, these systems were built for realtime data serving and not low-latency warehousing and analytics. Hence, their design choices are optimized for data freshness, unlike for Kodiak, which is optimized for performance.

Druid [32] promises realtime data ingestion and ad-hoc analytics at scale. It does not rely on materialized views and evaluates its queries at real time. Instead, it relies on columnar storage as well as compression to enhance its scalability. More importantly, it tries to keep all its data in

memory for faster access. It has the advantage over Kodiak of not being limited by available materialized views that it has to fall back to a much slower MapReduce-based system like Cheetah, if no view is available to serve the query. However, Druid has the fundamental scalability problem of relying on memory. For peta-byte-scale workloads having hundreds of thousands of expensive queries per day, this can become prohibitively expensive. Moreover, even if compressed and stored in memory, scanning a peta-byte-scale dataset is fundamentally expensive.

Elmeleegy *et al.* [12] presented an overview of Turn's Data Management Platform (DMP). The paper's focus though was data models used and ingestion techniques and management opposite to this paper, where the primary focus is on optimizing query performance. Song *et al.* [26] showed Turn's ETL pipeline and performance data population. However, this paper focuses on data storage with optimized query performance.

Finally, view maintenance in general has received a lot of attention in the literature in the past [8, 23, 18, 16, 14, 24]. More recently, Agrawal *et al.* [1] have worked on view maintenance for very large scale databases. In their work, they adopt a more eager approach to view maintenance than Kodiak, where updates to base tables are either reflected immediately or shortly to their corresponding views. Kodiak takes a different approach, where updates are batched into large batches and affected shards of the view are recomputed to include these updates. This comes at the expense of views' data freshness. Fortunately, degraded freshness is acceptable for our application. Further, even though recomputing entire shards is often wasteful as it includes redundant work because many records in the view do not change between maintenance iterations, batching updates and bulk loading mitigates this problem. In addition, Kodiak's scalability can handle any extra load. In return, our approach provides snapshot-consistency required for our application. This also significantly simplifies fault tolerance as updates do not need to be tracked at the record level.

6. CONCLUSIONS

This paper presents Turn's analytics platform for online advertising performance data – Kodiak. The platform is horizontally scalable enabling it to efficiently handle petabytes of data. It is designed for high performance and very low latency response time.

Kodiak's current version is used in production for over a year. It serves over 200,000 queries every day. These are complex high-dimensional aggregation queries. Still, Kodiak achieves very good performance with a median query latency of 8 ms and a 99 percentile latency of 252 ms.

It achieves this performance by leveraging materialized views. Views are user defined and system maintained. The system has the intelligence to select the best view for each query it receives. Kodiak trades data freshness for better performance, which complies with Turn's use case. Specifically, views are updated lazily in batches to reduce the maintenance cost.

Kodiak is fault tolerant. It supports Turn's web portal providing analytics to its customers. Hence, it is expected to have high availability. Consequently, it relies on replication to tolerate failures.

7. ACKNOWLEDGMENTS

We would like to thank the reviewers for their valuable feedback. We also thank the rest of our team members: Lijie Heng, Haoran Li, Brian Peltz, David Rubin, Arjun Satish, Pooya Shareghi, Thomas Shiou, Serkan Uzunbaz, Chuck Zhang, and Margaret Zhang for their help and support.

8. REFERENCES

[1] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan. Asynchronous view maintenance for vlsd databases. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 179–192, New York, NY, USA, 2009. ACM.

[2] HBase: the Hadoop database. `http:///hbase.apache.org/`.

[3] The Apache Cassandra database. `http:///cassandra.apache.org/`.

[4] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.

[5] A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A modern, open-source sql engine for hadoop. In *CIDR*, 2015.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Computer Systems*, 26(2), 2008.

[7] S. Chen. Cheetah: A high performance, custom data warehouse on top of mapreduce. *Proc. VLDB Endow.*, 3(1-2):1459–1468, Sept. 2010.

[8] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 469–480, New York, NY, USA, 1996. ACM.

[9] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

[10] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[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 *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[12] H. Elmeleegy, Y. Li, Y. Qi, P. Wilmot, M. Wu, S. Kolay, A. Dasdan, and S. Chen. Overview of turn data management platform for digital advertising. *PVLDB*, 6(11):1138–1149, 2013.

[13] K. Elmeleegy. Piranha: Optimizing short jobs in hadoop. *Proc. VLDB Endow.*, 6(11):985–996, Aug. 2013.

[14] A. Gupta and I. S. Mumick. Materialized views. chapter Maintenance of Materialized Views: Problems, Techniques, and Applications, pages 145–157. MIT Press, Cambridge, MA, USA, 1999.

[15] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIXATC'10: Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, pages 11–11, 2010.

[16] K. Y. Lee and M. H. Kim. Efficient incremental maintenance of data cubes. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 823–833. VLDB Endowment, 2006.

[17] C. Mohan. History repeats itself: Sensible and nonsensql aspects of the nosql hoopla. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 11–16, New York, NY, USA, 2013. ACM.

[18] I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of data cubes and summary tables in a warehouse. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, pages 100–111, New York, NY, USA, 1997. ACM.

[19] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *Proc. ACM SIGMOD*, 2008.

[20] Oracle. Automatic Storage Management. `http://docs.oracle.com/cd/E11882_01/server.112/e18951/asmcon.htm`.

[21] Oracle Clusterware. `http://www.oracle.com/technetwork/database/database-technologies/clusterware/overview/index.html`.

[22] L. Qiao, K. Surlaker, S. Das, T. Quiggle, B. Schulman, B. Ghosh, A. Curtis, O. Seeliger, Z. Zhang, A. Auradar, C. Beaver, G. Brandt, M. Gandhi, K. Gopalakrishna, W. Ip, S. Jgadish, S. Lu, A. Pachev, A. Ramesh, A. Sebastian, R. Shanbhag, S. Subramaniam, Y. Sun, S. Topiwala, C. Tran, J. Westerman, and D. Zhang. On brewing fresh espresso: Linkedin's distributed data serving platform. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1135–1146, New York, NY, USA, 2013. ACM.

[23] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 447–458, New York, NY, USA, 1996. ACM.

[24] K. Salem, K. Beyer, B. Lindsay, and R. Cochrane. How to roll a join: Asynchronous incremental view maintenance. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 129–140, New York, NY, USA, 2000. ACM.

[25] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A distributed sql database that scales. In *VLDB*, 2013.

[26] B. Song, S. Liu, S. Kolay, and L. Lo. Antsboa: A new time series pipeline for big data processing, analyzing and querying in online advertising application. In *First IEEE International Conference on Big Data Computing Service and Applications, BigDataService 2015, Redwood City, CA, USA, March 30 - April 2, 2015*, pages 223–232, 2015.

[27] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, Aug. 2009.

[28] M. Traverso. Presto: Interacting with petabytes of data at Facebook. `https://www.facebook.com/notes/facebook-engineering/presto-interacting-with-petabytes-of-data-at-facebook/10151786197628920`.

[29] Big Data Benchmark - AMPLab. `https://amplab.cs.berkeley.edu/benchmark`.

[30] VMware, Inc. Tungsten Replicator 3.0 Manual. Technical report, VMware, Inc, 2015.

[31] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 13–24, New York, NY, USA, 2013. ACM.

[32] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli. Druid: A real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 157–168, New York, NY, USA, 2014. ACM.