Pinot: Realtime OLAP for 530 Million Users

Jean-François Im
LinkedIn Corp.
Mountain View, California
jfim@linkedin.com

Kishore Gopalakrishna
LinkedIn Corp.
Mountain View, California
kgopalak@linkedin.com

Subbu Subramaniam
LinkedIn Corp.
Mountain View, California
ssubrama@linkedin.com

Mayank Shrivastava
LinkedIn Corp.
Mountain View, California
mshrivas@linkedin.com

Adwait Tumbde
LinkedIn Corp.
Mountain View, California
atumbde@linkedin.com

Xiaotian Jiang
LinkedIn Corp.
Mountain View, California
xajiang@linkedin.com

Jennifer Dai
LinkedIn Corp.
Mountain View, California
jdai@linkedin.com

Seunghyun Lee
LinkedIn Corp.
Mountain View, California
snlee@linkedin.com

Neha Pawar
LinkedIn Corp.
Mountain View, California
npawar@linkedin.com

Jialiang Li
LinkedIn Corp.
Mountain View, California
jlli@linkedin.com

Ravi Aringunram
LinkedIn Corp.
Mountain View, California
raringun@linkedin.com

ABSTRACT

Modern users demand analytical features on fresh, real time data. Offering these analytical features to hundreds of millions of users is a relevant problem encountered by many large scale web companies.

Relational databases and key-value stores can be scaled to provide point lookups for a large number of users but fall apart at the combination of high ingest rates, high query rates at low latency for analytical queries. Online analytical databases typically rely on bulk data loads and are not typically built to handle nonstop operation in demanding web environments. Offline analytical systems have high throughput but do not offer low query latencies nor can scale to serving tens of thousands of queries per second.

We present Pinot, a single system used in production at Linkedin that can serve tens of thousands of analytical queries per second, offers near-realtime data ingestion from streaming data sources, and handles the operational requirements of large web properties. We also provide a performance comparison with Druid, a system similar to Pinot.

CCS CONCEPTS

• **Information systems → Relational parallel and distributed DBMSs**; • **Software and its engineering** → *Cloud computing*;

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'18, June 10–15, 2018, Houston, TX, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3190661

ACM Reference Format:
Jean-François Im, Kishore Gopalakrishna, Subbu Subramaniam, Mayank Shrivastava, Adwait Tumbde, Xiaotian Jiang, Jennifer Dai, Seunghyun Lee, Neha Pawar, Jialiang Li, and Ravi Aringunram. 2018. Pinot: Realtime OLAP for 530 Million Users. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3183713.3190661

1 INTRODUCTION

Modern web companies generate large amounts of data, and increasingly sophisticated end users demand to be able to analyze ever growing volumes of data. Doing so at scale, with interactive-level performance requires sophisticated solutions to deliver the responsiveness that users have come to expect.

We postulate that the key requirements for a scalable near-realtime OLAP service are as follows:

Fast, interactive-level performance Users are not ready to wait for extended periods of time for query results, as this breaks the tight interaction loops needed to properly explore data;

Scalability A scalable service should provide near-linear scaling and fault tolerance to handle the demanding operational requirements of large scale web scale deployments in order to accomodate high numbers of concurrent queries while ingesting large amounts of data in a near-realtime fashion;

Cost-effectiveness As data volumes and query rates keep on increasing, the costs to serve user requests cannot grow unbounded, often requiring colocation of different use cases;

Low data ingestion latency Users expect to be able to query data points that have been added recently in a near-realtime fashion without having to wait for batch jobs to load data;

Flexibility Users demand the ability to drill down arbitrarily without being bound by pre-aggregated results as well as introduce new fixed query patterns in production without downtime;

Fault tolerance System failures should cause a graceful degradation for end users;

Uninterrupted operation The service should operate continuously, without downtime for upgrades or schema changes;

Cloud-friendly architecture The service should be easily deployable within the constraints of commercially available cloud services.

Interactive-level responsiveness is an important problem, being identified as a key challenge by various authors [8, 18]. Approaches such as MapReduce [10] and Spark [31] have adequate throughput but "their high latency and lack of online processing limit fluent interaction [15]." Furthermore, sophisticated users expect responsive dashboards and complex data visualizations that allow quick drill downs [16].

At web scale, scalability is a critical concern; any solution that does not offer near-linear scaling will eventually need to be replaced when the requirements exceed the scaling capacity. Most large-scale distributed data systems satisfy this near-linear scaling capability and Pinot is no exception to this.

Cost effectiveness is also a key concern in a scalable query engine. While technically it is possible to lower query latency and increase throughput by "throwing hardware at the problem", doing so quickly becomes prohibitively expensive when operating at large scale. Performance is intimately tied to cost effectiveness; hardware resources that are currently processing a query are unavailable for other queries, by definition. Thus, increasing performance also has the side effect of improving cost effectiveness.

Ingestion latency is another important facet of analytics. Many commercially available systems for analytics cannot handle single row loads and rely on *bulk loads* for data ingestion. This has the side effect of increasing the time from a business event happening and it being detectable by analysts.

An analytical system also needs to be flexible; limiting users to predefined combinations of dimensions when drilling down or otherwise preventing users from accessing record-level details hampers the data analysts in their daily work, causing them to query other systems to get more granular information. It also needs to be reconfigurable in order to allow for changing requirements from users.

Furthermore, different use cases for near-realtime OLAP systems have different complexity and throughput requirements. Dashboards for millions of end users might be limited to one or two facets due to the large amount of incoming queries, while lower throughput use cases will typically feature more complex drill down features.

Finally, fault tolerance and continuous operation are required to satisfy the demands of today's continuously operating web properties; as these web properties are available globally, there is no good window for system maintenance downtime.

Given these criteria, we present Pinot, a single system that serves near-realtime analytics for all Linkedin users, spanning from simple high throughput queries for large numbers of end-users such as newsfeed customization and "Who viewed my profile" to more complicated dashboards for advertiser analytics and internal analytics. We present our findings from operating Pinot at scale, and compare the performance and scalability of the different indexing techniques implemented in Pinot on production workloads. We also compare the performance of Pinot with Druid [30], an analytical system with an architecture similar to Pinot.

2 RELATED WORK

We now survey different approaches to online analytics.

Traditional row-oriented RDBMS can process OLAP-style queries, albeit inefficiently. This has the advantage of allowing analytical queries on the same system used for OLTP and avoiding to incur the maintenance costs of multiple systems. At smaller web companies, one oft-encountered configuration is to build a read replica of the OLTP database, which is then used for analytical purposes. However, as data volumes increase, the cost of maintaining per-column indexes limits ingest rates for such setups. Furthermore, as OLTP databases are often normalized while OLAP analysis is significantly faster using star schemas, snowflake schemas, or fully denormalized tables, the performance of analytics on such systems eventually becomes unacceptable.

Column stores, such as MonetDB [5], C-Store [26], Vertica [21], and many others offer significant performance improvements over row-oriented stores for analytical queries. As analytical queries tend to scan large amounts of data for a subset of all columns, column stores improve the query performance by avoiding transferring data that is not used to compute the query result, and allow for compression and various optimizations [2, 14] that are not possible in row-oriented data stores. However, column stores do not do very well for certain operations, such as single-row inserts, updates, and point lookups.

Some newer databases, such as SAP HANA [12], DB2 BLU [25], Oracle Database in-memory [20], and MemSQL [9], integrate both row-oriented and columnar execution within the same database. In these hybrid transactional/analytical processing databases, tables can be row or column oriented — or even both — allowing users to mitigate the drawbacks of either orientation by using the optimal data orientation for their needs.

At large scale, "offline" approaches to OLAP are also used. Systems such as Hive [27], Impala [4], and Presto [1] offer distributed query execution on large data sets. Such systems do not keep user data resident in memory between queries and instead rely on operating system caches to speed up repeated queries on the same data set; their execution model is to execute queries on a set of distributed worker nodes. As shown in the performance evaluation of Spark [31], the cost of loading data from storage for each query is significant. Furthermore, as such systems do not keep data resident in memory, they cannot handle data that has not been written to durable distributed storage; in practice, this translates to a data availability gap between transactional systems and analytical systems. Finally, such systems have a relatively high per-query set up time (from hundreds of milliseconds to several dozens of seconds), precluding the execution of tens of thousands of queries per second.

There are also large performance gains to be had by building more specialized systems. For example, Figure 6 of [17] shows how a specialized system can have order of magnitude improvements in performance over more general approaches. One way to improve the performance of OLAP system is to preaggregate data into cubes, then perform query execution on the preaggregated data. As each cube can contain an unbounded number of rows, this can offer performance improvements of several orders of magnitude. At large scale, the cubes can be stored in distributed key-value stores [28, 32].

However, these performance improvements come at the expense of query flexibility; queries that contain dimensions for which there is no preexisting aggregate cannot be executed. This limits the range of queries to certain combinations of dimensions. Furthermore, some resolution is lost in the preaggregation, so filtering based on timestamps with fine granularity or computing exact values for summary statistics that require the original data (median, distinct count, etc.) are not possible.

Another example of a specialized system is Druid [30]. Similar to Pinot, Druid is an *asynchronous low latency ingestion analytical store*. Unlike transactional systems, data is loaded asynchronously in Druid by writing it into a queuing system (Kafka is used for both Druid and Pinot). The asynchronous loading allows the producer of events to quickly write desired business events to the queuing system without waiting for a transaction to complete. This means that critical front-end transactional processing is not blocked on back-end analytical system availability. Ingestion in Druid is also low latency, as indexing of events happens shortly after being written to Kafka as opposed to being bulk loaded periodically.

Both Druid and Pinot share similar architectural choices; query execution is distributed, data is ingested asynchronously from streaming sources, and both trade off strong consistency for eventual timeline consistency. However, unlike Druid, Pinot has been optimized for handling both high throughput serving of simple analytical queries and more complex analytical queries at lower rates. Furthermore, the integration of additional specialized data structures and certain optimizations, as described in section 4, allows for high throughput serving of low complexity aggregation queries at tens of thousands of queries per second in production environments.

Table 1: A comparison of the techniques for OLAP and their applicability to large scale serving.

Technique	Fast ingest and indexing	High query rate	Query flexibility	Query latency
RDBMS	Not typically	Yes	High	Low/moderate
KV stores	Yes	Yes	None	Low
Online OLAP	No	Not typically	High	Low/moderate"
"Offline" OLAP	No	No	High	High
Druid	Yes	No	Moderate	Low/moderate
Pinot	Yes	Yes	Moderate	Low

3 ARCHITECTURE

Pinot is a scalable distributed OLAP data store developed at LinkedIn to deliver real time analytics with low latency. Pinot is optimized for analytical use cases on immutable append-only data and offers data freshness that is on the order of a few seconds. We have been running Pinot in production at LinkedIn for many years across hundreds of servers and tables processing thousands of queries per second. Importantly, Pinot is used to power customer facing applications such as "Who viewed my profile" (WVMP) and newsfeed customization which require very low latency as well as internal business analyst dashboards where users want to slice and dice data.

At Linkedin, business events are published in Kafka streams and are ETL'ed onto HDFS. Pinot supports near-realtime data ingestion by reading events directly from Kafka [19] as well as data pushes from offline systems like Hadoop. As such, Pinot follows the lambda architecture [23], transparently merging streaming data from Kafka and offline data from Hadoop. As data on Hadoop is a global view of a single hour or day of data as opposed to a direct stream of events, it allows for the generation of more optimal segments and aggregation of records across the time window.

3.1 Data and Query Model

Just like typical databases, data in Pinot consists of records in tables. Tables have a fixed schema composed of multiple columns. Supported data types are integers of various lengths, floating point numbers, strings and booleans. Arrays of the previous types are also supported. Each column can be either a dimension or a metric.

Pinot also supports a special timestamp dimension column called a *time column*. The time column is used when merging offline and realtime data as explained in section 3.3.3 and for managing automatic data expiration.

Tables are composed of *segments*, which are collections of records. A typical Pinot segment might have a few dozen million records and tables can have tens of thousands of segments. Segments are replicated, ensuring data availability. Data in segments is immutable, although segments themselves can be replaced with a newer version; this allows for updates and corrections to existing data.

Data orientation in Pinot segments is columnar. Various encoding strategies are used to minimize the data size, including dictionary encoding and bit packing of values. Inverted indexes are also supported. A typical segment is a few hundred megabytes up to a few gigabytes.

Querying in Pinot is done through PQL, a subset of SQL. PQL is modeled around SQL and supports selection, projection, aggregations, and top-n queries, but does not support joins or nested queries. PQL does not offer any DDL nor record-level creation, updates or deletion.

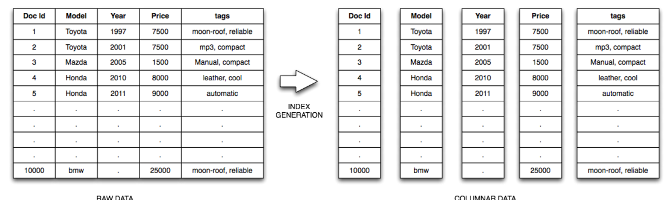

Figure 1: Pinot Segment

3.2 Components

Pinot has four main components for data storage, data management, and query processing: controllers, brokers, servers, and minions. Additionally, Pinot depends on two external services: Zookeeper and a persistent object store. Pinot uses Apache Helix [13] for cluster management. Apache Helix is a generic cluster management framework that manages partitions and replicas in a distributed system.

Servers are the main component responsible for hosting segments and processing queries on those segments. A segment is stored as a directory in the UNIX filesystem consisting of a segment metadata file and an index file. The segment metadata provides information about the set of columns in the segment, their type, cardinality, encoding, various statistics, and the indexes available for that column. An *index file* stores indexes for the all the columns. This file is append-only which allows the server to create inverted indexes on demand. Servers have a pluggable architecture that supports loading columnar indexes from different storage formats as well as generating synthetic columns at runtime. This can be easily extended to read data from distributed filesystems like HDFS or S3. We maintain multiple replicas of a segment within a datacenter for higher availability and query throughput. All replicas participate in query processing.

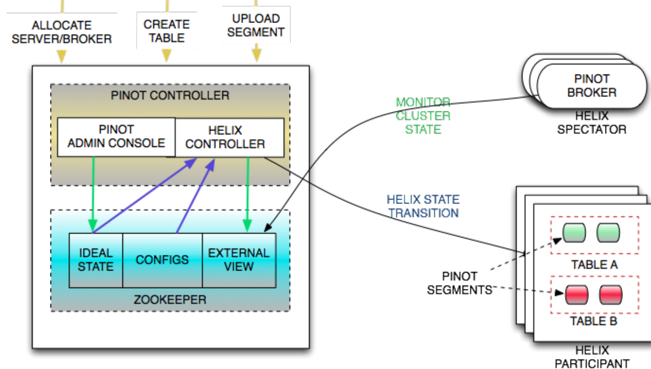

Figure 2: Pinot Cluster Management

Controllers are responsible for maintaining an authoritative mapping of segments to servers using a configurable strategy. Controllers own this mapping and trigger changes to it on operator requests or in response to the changes in server availability. Additionally, controllers support various administrative tasks such as listing, adding, or deleting tables and segments. Tables can be configured to have a retention interval after which segments past the retention period are garbage collected by the controller. All the metadata and mapping of segments to servers is managed using Apache Helix. For fault tolerance, we run three controller instances in each datacenter with a single master; non-leader controllers are mostly idle. Controller mastership is managed by Apache Helix.

Brokers route incoming queries to appropriate server instances, collect partial query responses, merge them into a final result, which is then sent back to the client. Pinot clients send their queries to brokers over HTTP, allowing for load balancers to be placed in front of the pool of brokers.

Minions are responsible for running compute-intensive maintenance tasks. Minions execute tasks assigned to them by the controllers' job scheduling system. The task management and scheduling is extensible to add new job and schedule types in order to satisfy evolving business requirements.

An example of a task that is run on the minions is data purging. Linkedin must sometimes purge member-specific data in order to comply with various legal requirements. As data in Pinot is immutable, a routine job is scheduled to download segments, expunge the unwanted records, rewrite and reindex the segments before finally uploading them back into Pinot, replacing the previous segments.

Zookeeper is used as a persistent metadata store and as the communication mechanism between nodes in the cluster. All information about the cluster state, segment assignment and metadata is stored in Zookeeper though Helix. Segment data itself is stored in the persistent object store. At Linkedin, Pinot uses a local NFS mountpoint for data storage but we have also used Azure Disk storage when running outside of Linkedin's datacenters.

3.3 Common Operations

We now explain how common operations are implemented in Pinot.

3.3.1 Segment Load. Helix uses state machines to model the cluster state; each resource in the cluster has its own current state and desired cluster state. When either state changes, the appropriate state transitions are sent to the respective nodes to be executed.

Pinot uses a simple state machine for segment management, as shown in Figure 3. Initially, segments start in the OFFLINE state and Helix requests server nodes to process the OFFLINE to ONLINE transition. In order to handle the state transition, servers fetch the relevant segment from the object store, unpack it, load it, and make it available for query execution. Upon the completion of the state transition, the segment is marked as being in the ONLINE state in Helix.

For realtime data that is to be consumed from Kafka, a state transition happens from the OFFLINE to CONSUMING state. Upon processing this state transition, a Kafka consumer is created with a given start offset; all replicas for this particular segment start consuming from Kafka at the same location. A consensus protocol described in section 3.3.6 ensures that all replicas converge towards an exact copy of the segment.

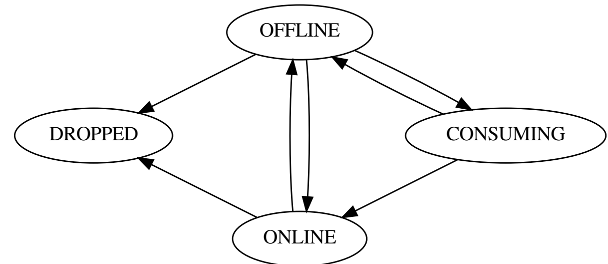

Figure 3: Pinot Segment State Machine

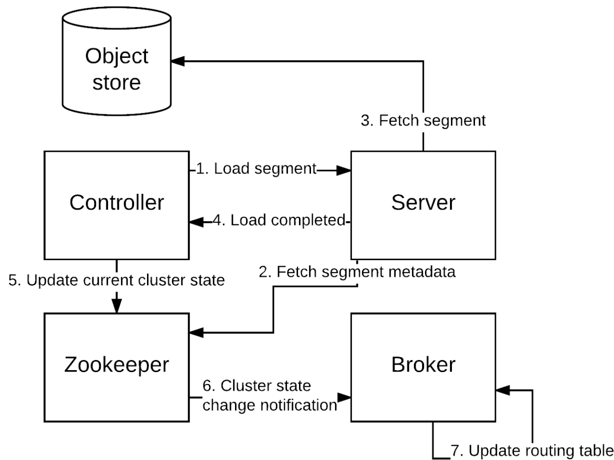

Figure 4: Pinot Segment Load

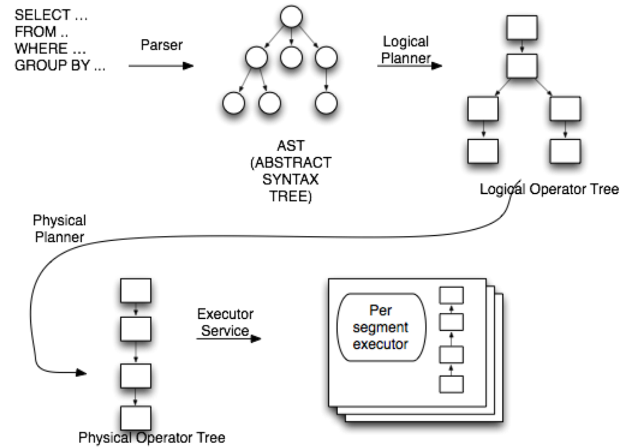

Figure 5: Query Planning Phases

3.3.2 Routing Table Update. When segments are loaded and unloaded, Helix updates the current cluster state. Brokers listen to changes to the cluster state and update their routing tables, a mapping between servers and available segments. This ensures that brokers are routing queries to replicas that are available as new replicas come online or are marked as unavailable. The process of routing table creation is described in more detail in section 4.4.

3.3.3 Query Processing. When a query arrives on a broker, several steps happen:

(1) The query is parsed and optimized
(2) A routing table for that particular table is picked at random
(3) All servers in the routing table are contacted and asked to process the query on a subset of segments in the table
(4) Servers generate logical and physical query plans based on index availability and column metadata
(5) The query plans are scheduled for execution
(6) Upon completion of all query plan executions, the results are gathered, merged and returned to the broker
(7) When all results are gathered from the servers, the partial per-server results are merged together. Errors or timeouts during processing cause the query result to be marked as partial, so that the client can choose to either display incomplete query results to the user or resubmit the query at a later time.
(8) The query result is returned to the client

Pinot supports dynamically merging data streams that come from offline and realtime systems. To do so, these *hybrid* tables contain data that overlaps temporally. Figure 6 shows that a hypothetical table with two segments per day might have overlapping data for August 1st and 2nd. When such a query arrives in Pinot, it is transparently rewritten into two queries; one query for the offline part, which queries data prior to the time boundary, and a second one for the realtime part, which queries data at or after the time boundary.

When both queries complete, the results are merged, allowing us to cheaply provide merging of offline and realtime data. In order for this scheme to work, hybrid tables require having a *time column* that is shared between the offline and realtime tables. In practice, we have not found this to be an onerous requirement, as most data written to streaming systems tend to have a temporal component.

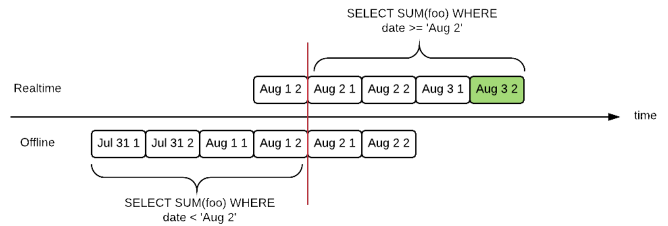

Figure 6: Hybrid Query Rewriting

3.3.4 Server-Side Query Execution. On the server-side, when a query is received, logical and physical query plans are generated. As available indexes and physical record layouts can be different between segments, query plans are generated on a per-segment basis. This allows Pinot to do certain optimizations for special cases, such as a predicate matching all values of a segment. Special query plans are also generated for queries that can be answered using segment metadata, such as obtaining the maximum value of a column without any predicates.

Physical operator selection is done based on an estimated execution cost and operators can be reordered in order to lower the overall cost of processing the query based on per-column statistics. The resulting query plans are then submitted for execution to the query execution scheduler. Query plans are processed in parallel.

3.3.5 Data Upload. To upload data, segments are uploaded to the controller using HTTP POST. When a controller receives a segment, it unpacks it to ensure its integrity, verifies that the segment size would not put the table over quota, writes the segment metadata in Zookeeper, then updates the desired cluster state by assigning the segment to be in the ONLINE state on the appropriate number of replicas. Updating the desired cluster state then triggers the segment load as described earlier.

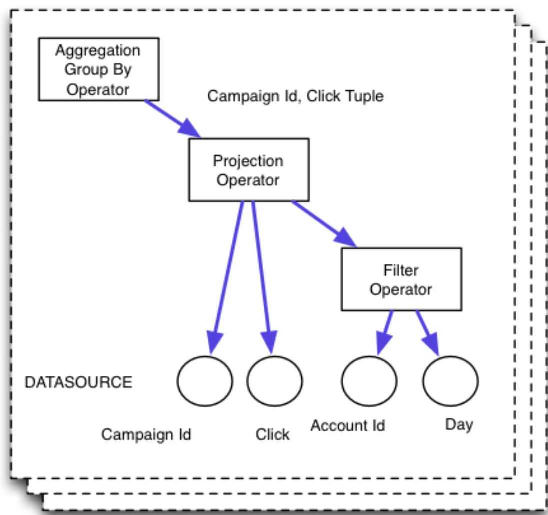

Figure 7: Query Planning Phases

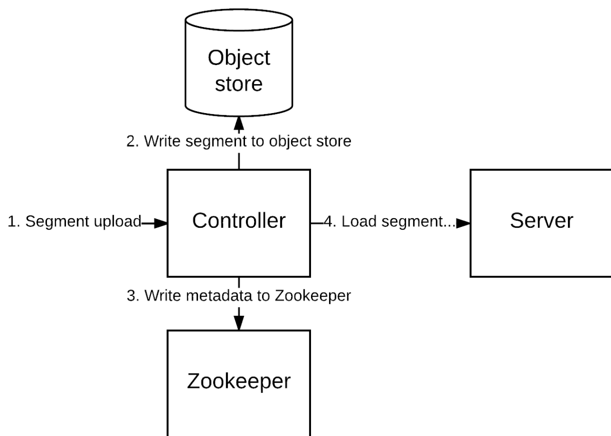

Figure 8: Pinot Data Upload

3.3.6 Realtime Segment Completion. In Pinot, realtime data consumption from Kafka happens on independent replicas. Each replica starts consuming from the same start offset and has the same end criteria for the realtime segment. When the end criteria is reached, the segment is flushed to disk and committed to the controller. As Kafka retains data only for a certain period of time, Pinot supports flushing segments after a configurable number of records and after a configurable amount of time.

Independent consumers consuming from the same Kafka offset and partition for the exact same number of records will consume the same exact data; however, two consumers consuming for a certain amount of time based on their local clock will likely diverge. As such, Pinot has a segment completion protocol that ensures that independent replicas have a consensus on what the contents of the final segment should be.

When a segment completes consuming, the server starts polling the leader controller for instructions and gives its current Kafka offset. The controller then returns a single instruction to the server. Possible instructions are:

HOLD Instructs the server to do nothing and poll at a later time

DISCARD Instructs the server to discard its local data and fetch an authorative copy from the controller; this happens if another replica has already successfully committed a different version of the segment

CATCHUP Instructs the server to consume up to a given Kafka offset, then start polling again

KEEP Instructs the server to flush the current segment to disk and load it; this happens if the offset the server is at is exactly the same as the one in the committed copy

COMMIT Instructs the server to flush the current segment to disk and attempt to commit it; if the commit fails, resume polling, otherwise, load the segment

NOTLEADER Instructs the server to look up the current cluster leader as this controller is not currently the cluster leader, then start polling again

Replies by the controller are managed by a state machine that waits until enough replicas have contacted the controller or enough time has passed since the first poll to determine a replica to be the committer. The state machine attempts to get all replicas to catch up to the largest offset of all replicas and picks one of the replicas with the largest offset to be the committer. On controller failure, a new blank state machine is started on the new leader controller; this only delays the segment commit, but otherwise has no effect on correctness.

This approach minimizes network transfers while ensuring all replicas have identical data when segments are flushed.

3.4 Cloud-Friendly Architecture

Pinot has been specifically designed to be able to run on cloud infrastructure. Commercially available cloud infrastructure providers provide the two important ingredients required for Pinot execution: a compute substrate with local ephemeral storage and a durable object storage system.

As such, Pinot has been designed as a share-nothing architecture with stateless instances. In Pinot, all persistent data is stored in the durable object storage system and system metadata is stored in Zookeeper; local storage is only used as a cache and can be recreated by pulling data from the durable object storage or from Kafka. As such, any node can be removed at any time and replaced by a blank one without any issues.

Furthermore, all user-accessible operations for Pinot are done through HTTP, allowing users to leverage existing battle-tested load balancers — such as HAProxy or nginx — or client-side software load balancers like Linkedin's D2.

This cloud-friendly architecture has allowed us to trivially port Pinot to be ran on off the shelf container execution services with

only the code changes required to interface with the cloud provider's object storage system. Such an architecture allows for easy deployment and scaling using container managers such as Kubernetes.

4 SCALING PINOT

Several features of Pinot were essential to get acceptable performance at scale. We cover these features and explain how they enable Pinot to serve analytical queries for Linkedin's users.

4.1 Query Execution

Pinot's query execution model has been designed to accommodate new operators and query shapes. For example, the initial version of Pinot did not support metadata-only queries for queries such as `SELECT COUNT(*)`. Adding support for such queries involved a few changes to the query planner and adding a new metadata-based physical operator, but did not require any architectural changes.

Pinot's physical operators are specialized for each data representation; there are operators for each different data encoding. This flexibility allows us to add new index types and specialized data structures for query optimization. As we can reindex data on the fly on servers themselves or through the minion subsystem, it is possible for us to deploy new index types and encodings without users of Pinot being aware of such changes.

4.2 Indexing and Physical Record Storage

Similar to Druid, we support bitmap-based inverted indexes. However, we have observed that physically reordering the data based on primary and secondary columns allows certain types of queries to run significantly faster.

For example, for the "Who viewed my profile" feature of the Linkedin website, all queries have a filter on the `vieweeId` column. As such, physically reordering the records based on the `vieweeId` column ensures that for any given query, only a contiguous section of the column needs to be considered, allowing Pinot to store only the start and end index into the column position for any given `vieweeId`. This adjacency also makes it possible to use vectorized query execution in the case where there are no other query predicates.

In the case where vectorized query execution is not possible, we have observed that falling back to iterator-style scan query execution on a range of the column leads to better query performance than trying to perform bitmap operations on large bitmap indexes.

As such, when creating physical filter operators, the ones operating on the physically sorted column are executed first and pass on their column range to subsequent operators. This causes subsequent operators to only evaluate part of the column, greatly improving performance.

4.3 Iceberg Queries

An important class of queries are *iceberg queries* [11], where only aggregates that satisfy a certain minimum criteria are returned. For example, an analyst might be interested in knowing which countries contribute the most page views for a given page, but not the entire list of all countries that visited the page; for such a query, returning the countries that exceed a minimum page view threshold is sufficient to answer the question the analyst had. This is especially important in data sets which have a long tail distribution and analysts that are mostly concerned with "moving the needle" on key metrics. Such queries happen frequently in dashboarding use cases.

Iceberg cubing [3] expands on OLAP cubes by adapting them to answer iceberg queries. Further work on iceberg cubing brought several advances, such as *star-cubing* [29], which improves the iceberg cubing technique by making it more efficient to compute in comparison to other iceberg cubing approaches for most cases. In star-cubing, a pruned hierarchical structure of nodes called a *star-tree* is constructed and can be efficiently traversed to answer queries.

Star-trees consist of nodes of preaggregated records; each level of the tree contains nodes that satisfy the iceberg condition for a given dimension and a *star-node* which represents all data for this particular level. Navigating the tree allows answering queries with multiple predicates. For example, Figure 9 shows a simple query to obtain the sum of all impressions with a simple predicate; to answer the query, each level of the tree is navigated until finding the node that contains the aggregated data that answers this specific query. Figure 10 shows a more complex query with an `or` predicate for which multiple navigations are required to answer the query.

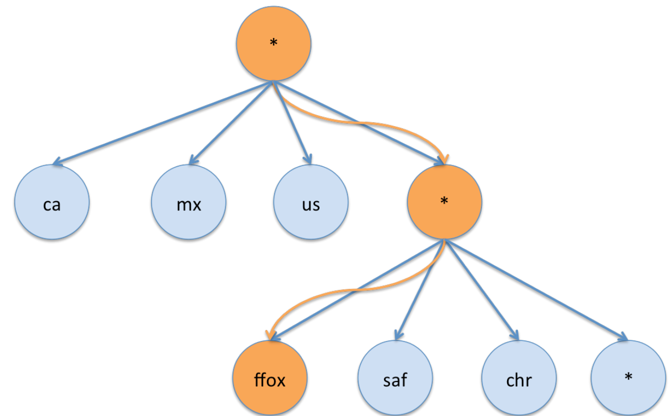

Figure 9: `select sum(Impressions) from Table where Browser = 'firefox'`

We have implemented star-trees in Pinot and routinely use them to speed up analytical queries for our internal data analytics tools. As Pinot determines which physical query execution nodes can be applied given the indexes available, if a user specifies a query that can be optimized by using the star-tree structure, we transparently use it to return pre-aggregated values; otherwise, query execution runs on the original unaggregated data.

4.4 Query Routing and Partitioning

In Pinot, for unpartitioned tables, we pre-generate a routing table, which is a list of mappings between servers and their subset of segments to be processed when a query is executed. Formally, given a set of segments $U = \{S_1, S_2, ..., S_n\}$ and a collection A of m sets that represent the segments assigned to each server, generating a

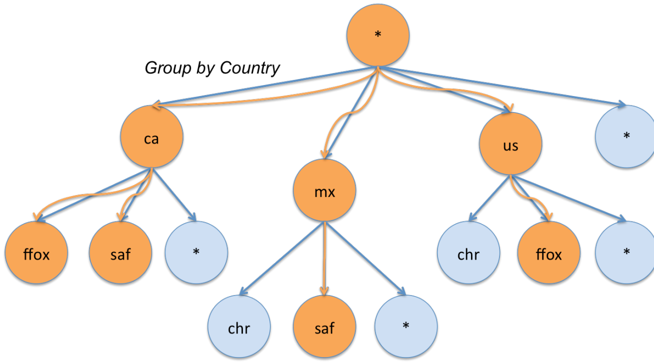

Figure 10: `select sum(Impressions) from Table where Browser = ’firefox’ or Browser = ’safari’ group by Country`

single routing table entry is the act of picking a collection of subsets of the elements of A such that the union of the sub-sets in the collection is equivalent to U.

Pinot supports various query routing options, which were found to be necessary at scale. The default query routing strategy in Pinot is a *balanced* strategy that simply divides all the segments contained in a table in an equal fashion across all available servers. In other words, when a query is processed, all servers are contacted and given a query to execute on their share of the segments to process.

The balanced strategy works well for small and medium sized clusters, but quickly becomes impractical for larger clusters. Intuitively, we can guess that the larger the cluster, the more likely it is that a single host in the cluster will be unavailable or have issues that slow down query processing. Figure 14 and 16 of [24] empirically show that such stragglers exist even in other systems.

As such, Pinot has a special routing strategy for large clusters that minimizes the number of hosts contacted in the cluster for any given query; this minimizes the adverse impact of any given misbehaving host and reduces tail latency for larger clusters.

Since picking the exact minimal subset of A such that U is covered is a NP-hard problem, we have implemented a random greedy strategy that produces an approximately minimal subset that also ensures a balanced load across all servers. Algorithms 1 and 2 explain how routing tables are generated and selected during the large cluster routing table generation. During the large cluster routing table generation, many routing tables are generated by taking a random subset of servers and adding additional servers until U is completely covered; segments are then assigned as evenly as possible amongst the servers selected. For each routing table generated, a metric is used to determine the routing table's fitness — empirical testing has shown that the variance of the number of segments assigned per server works well — and the routing tables that have the lowest metrics are kept.

Pinot also supports partitioned tables, where data is partitioned according to a partition function. When a table is partitioned, the router does not generate routing tables but rather routes queries only to the servers that contain relevant segments given the query filters. Since Pinot supports realtime ingestion of data from Kafka, Pinot includes a partition function that matches the behavior of the Kafka partition function, allowing for Pinot offline data to be partitioned in the same way as the realtime data.

Algorithm 1 Routing table generation

▷ S a set of segments for this table
▷ I a set of instances assigned for this table
▷ T the target server count per query
▷ S_{orphan} a set of segments with no servers associated
▷ I_{used} a set of instances in use
▷ IS a map of instances to a list of segments served by the instance
▷ SI a map of segments to a list of instances that serves this segment
▷ Q_{si} a priority queue of segments and potential instances list, sorted in ascending order of the length of the instance list

procedure GenerateRoutingTable
  $S_{orphan} \leftarrow S$ ▷ Initially, all segments have no associated instance
  $I_{used} \leftarrow \varnothing$
  if $length(I) \leq T$ **then**
    $I_{used} \leftarrow I$ ▷ If there are less than T instances, use all instances
    $S_{orphan} \leftarrow \varnothing$
  else
    while $length(I_{used}) \leq T$ **do** ▷ Pick T random instances
      $I_{random} \leftarrow \text{PickRandom}(I)$
      $I_{used} \leftarrow I_{used} \cup \{I_{random}\}$
      $S_{orphan} \leftarrow S_{orphan} - IS.get(I_{random})$
    end while
  end if
  while $S_{orphan} \neq \varnothing$ **do** ▷ Add servers to serve orphan segments
    $I_{random} \leftarrow \text{PickRandom}(SI.get(S_{orphan}.first))$
    $I_{used} \leftarrow I_{used} \cup \{I_{random}\}$
    $S_{orphan} \leftarrow S_{orphan} - IS.get(I_{random})$
  end while
  $Q_{si} \leftarrow \varnothing$
  for $S_{current} \leftarrow S$ **do**
    $I_{seg} \leftarrow SI.get(S_{current}) \cap I_{used}$ ▷ Get instances for this segment
    $Q_{si}.put(S_{current}, I_{seg})$
  end for
  $R \leftarrow \varnothing$
  while $Q_{si} \neq \varnothing$ **do** ▷ Iterate segments in ascending order of instances
    $(S_{current}, I_{seg}) \leftarrow Q_{si}.takeFirst()$
    $I_{picked} \leftarrow \text{PickWeightedRandomReplica}(R, I_{seg})$
    $R.put(S_{current}, I_{picked})$
  end while
  return R
end procedure

Algorithm 2 Routing table selection

▷ H a max heap of routing tables and their associated metric
▷ C the target routing table count
▷ G the number of routing tables to generate

```
procedure FilterRoutingTables
    H ← ∅
    for i ← 1..C do
        R ← GenerateRoutingTable
        M ← ComputeMetric(R)
        H.put(R, M)
    end for
    for i ← C..G do
        R ← GenerateRoutingTable
        M ← ComputeMetric(R)
        (R_top, M_top) ← H.top()
        if M ≤ M_top then    ▷ New routing table better than the worst one?
            H.pop()    ▷ Yes, remove worst and add new one
            H.put(R, M)
        end if
    end for
end procedure
```

4.5 Multitenancy

For larger companies, having dedicated clusters on a per-use case basis eventually becomes problematic; at Linkedin, there are currently several thousand tables split between more than 50 tenants.

As such, Pinot supports multitenancy, with multiple tenants colocated on the same hardware. To prevent any given tenant from starving other tenants of query resources, a token bucket is used to distribute query resources on a per tenant basis. Each query deducts a number of tokens from its tenant's bucket that is proportional to the query execution time; when the bucket is empty, queries are enqueued to be processed whenever tokens are available again. The token bucket slowly refills over time, allowing for short transient spikes in query loads but preventing a misbehaving tenant from exhausting resources for other colocated tenants.

5 PINOT IN PRODUCTION

At Linkedin, Pinot runs on over 3000 geographically distributed hosts, serves over 1500 tables with over one million segments, for a total compressed data size of nearly 30 TiB (excluding data replication). Pinot's current production query rate across all data centers exceeds 50000 queries per second.

We now discuss the practical lessons learned while running Pinot at scale at Linkedin.

5.1 Use Case Types

At Linkedin, we have observed that users of Pinot tend to be split between two categories:

- Use cases with high throughput, low complexity queries for relatively simple analytical features like "who viewed my profile" and feed customization.
- Use cases with low query rates but more complex queries or larger data volumes, such as advertiser audience reach, self-service analytical tools or anomaly detection tools

The first type of use cases typically requires data to be present in main memory in order to serve the tens of thousands of queries per second required. These users tend to run a very small number of query patterns.

The latter type of use cases are typically colocated on hardware with NVMe storage, as the lower query rates and more lenient latency expectations make it possible to simply load the data on demand. These users typically have low query rates, but tend to have bursty spikes of queries when users request dashboards or start analyzing anomalies. For these use cases, colocation with other tenants is important to minimize the hardware footprint, as otherwise the hardware would be idle for a significant fraction of the time.

5.2 Operational Concerns

At Linkedin, Pinot is operated using a service model; teams developing user features use Pinot just like any other service while the Pinot team provides ongoing support and runs the service on a day-to-day basis. As the number of teams using Pinot increases over time, having dedicated staff to handle support and tuning on a per team basis would mean that the staffing requirements would increase over time. As such, we have designed Pinot to enable a self-service model as much as possible.

For example, Pinot allows changing schemas on the fly to add new columns without downtime. When a new column is added to an existing schema, it is automatically added with a default value on all previously existing segments and made available within a few minutes. We also parse the query logs and execution statistics on an ongoing basis in order to automatically add inverted indexes on columns where they would prove beneficial.

Replicating table configurations across multiple data centers and environments (testing and production) has routinely been a problem. Currently, our solution is to store table configurations in source control and synchronize them with Pinot on an ongoing basis through Pinot's REST API. This allows us to have an audit trail of changes and leverage search, validation, and code review tooling for all schema, index and configuration changes.

6 PERFORMANCE

We now evaluate the performance of Pinot in various scenarios, ranging from low throughput use cases to high throughput ones. As Druid is a system similar to Pinot, we also compare the performance of Druid and Pinot on those scenarios. As to ensure a realistic evaluation, data sets and queries were pulled from production systems; the data sets comprised the entirety of the data for each scenario evaluated while the queries were sampled to have tens of thousands of different queries in order to simulate a production environment. Realtime ingestion was disabled for both systems, due to the complexity of setting a repeatable environment shared between Pinot and Druid.

For both Pinot and Druid, nine hosts equipped with single socket Xeon® E5-2680 v3 processors running at 2.50 GHz are used to run the query processing. For Pinot, this means that the Pinot server is installed on these hosts; for Druid, the historical server and the

middle manager are running on these hosts, as recommended by the Druid documentation. Each host has 64 GiB of RAM installed and 2.8 TB of NVMe storage attached. The operating system used is Red Hat Enterprise Linux release 6.9 (Santiago).

For query distribution, three instances of the Pinot broker are running. The brokers are equipped with dual Xeon® E5-2630 v2 processors clocked at 2.60GHz and 64 GiB of RAM. One host runs RHEL release 6.5 while the other two run RHEL release 6.6. For Druid, the brokers are equipped with a single Xeon® E5-2680 v3 CPU clocked at 2.50GHz and 64 GiB of RAM. One broker runs RHEL 6.7 while the other two run RHEL 6.9. One broker host is also running the Druid coordinator and Druid overlord.

The first scenario in which we evaluate the performance of Pinot and Druid replicates the environment used for ad hoc reporting and anomaly detection on multidimensional key business metrics at Linkedin. As our anomaly detection system has a user-visible component, the query set for this scenario has both automatically generated queries used for monitoring as well as ad hoc queries based on users doing root cause analysis for detected anomalies; this means that Pinot needs to offer both high throughput for automated queries as well as low latency queries for interactive queries coming from users. Queries contain aggregations of metrics with a variable number of filtering predicates and grouping clauses, depending on the specific drill down requested by the user.

The data size for this scenario is 16 GB of data for Pinot and 13.8 GB of data for Druid. Figure 11 shows the latency of different indexing strategies as the query rate increases. On this dataset, the performance of Druid quickly becomes too high for interactive purposes. As expected, the performance of Pinot without indexes also drops out quickly. Adding inverted indices also increases the scalability of Pinot by a factor of two for this dataset, but the largest gain in scalability comes from integrating the star tree index structure described in section 4.3.

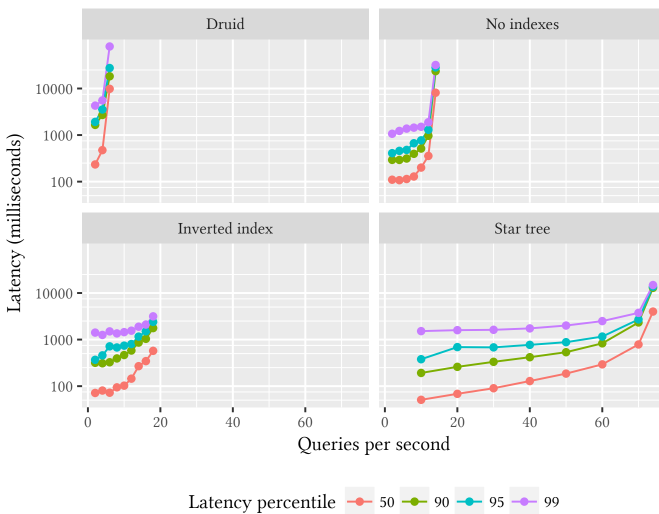

Figure 11: Comparison of indexing techniques on the anomaly detection dataset

The kernel density estimate plot in figure 12 shows the distribution of the latency of Druid and different indexing techniques implemented in Pinot as 10000 queries are executed sequentially. We can see that all systems have performance that is acceptable for user interaction. We can see that Druid has comparable performance with Pinot, when there are no indexes in Pinot; some queries execute faster in Druid, but Druid also has more queries that have high latency than Pinot without indexes. We can also see that adapted index types improve performance over unindexed data.

Figure 13 shows the distribution of the ratio of preaggregated records scanned during query execution using star tree versus the number of original unaggregated records. A ratio close to zero means that fewer aggregated records are used to process a query than execution over raw data, while a ratio close to one means that there are little gains from preaggregation. We can see that most queries execute on substantially fewer records than execution on raw, unaggregated data.

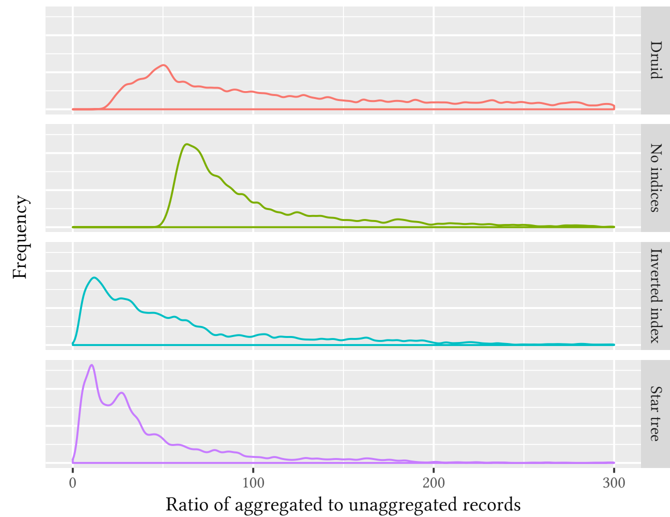

Figure 12: Distribution of query latency when running queries sequentially on the anomaly detection dataset

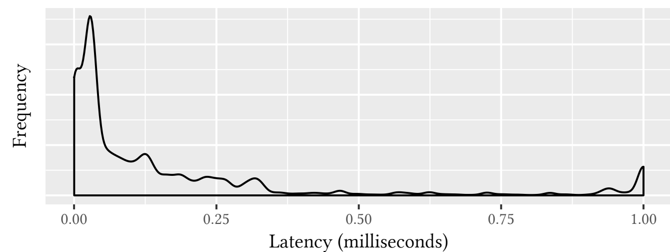

Figure 13: Distribution of the ratio of preaggregated records scanned during query execution using star tree versus the number of original unaggregated records

Pinot is also used to answer high selectivity analytical queries from end users. Examples of this are the various analytical tools available for end users that allow some limited analytics on who

viewed their published content as well as “who viewed my profile.” Queries for these scenarios are simple aggregations (sum of clicks/views, distinct count of viewers) with a few facets such as region, seniority or industry for a piece of shared content or a given user’s profile views.

The data size for Pinot is 300 GB and 1.2 TB for Druid. Figure 14 shows the performance of Druid and Pinot as the query rate increases. For this particular comparison, there are two major differences between Pinot and Druid: the generation of inverted indexes and the physical row ordering. In Druid, all dimension columns have an associated inverted index; as not all dimensions are used in filtering predicates, this leads to a larger on disk size for Druid over Pinot. A large part of the performance difference between Druid and Pinot in this comparison is due to the physical row ordering in Pinot, where data is sorted based on the shared item identifier.

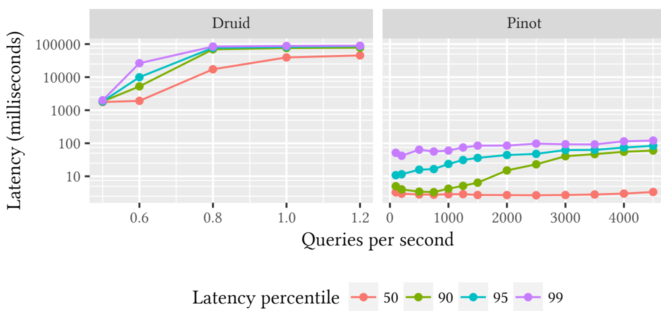

Figure 14: Comparison of Druid and Pinot on the “share analytics” dataset

As discussed in section 4.2, the physical ordering of records has a significant impact on scalability. Figure 15 illustrates the scalability difference between physically ordered records and bitmap-based inverted indexes in Pinot when running against the “who viewed my profile” dataset. Both Druid and Pinot use roaring bitmaps [6, 7] for their implementation of bitmap-based inverted indexes.

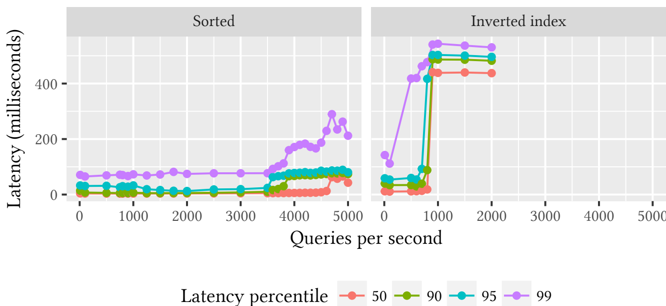

Figure 15: Comparison of indexing techniques on the “Who viewed my profile” dataset

Another interesting type of scenario for Pinot is illustrated by the implementation of impression discounting [22]. Impression discounting is the practice of tracking what items have been seen by a particular user and using this to personalize the display of items for that user; items that have been seen before by a user are “discounted” based on their interaction with these items so that ignored items are ranked lower for that user. This way, any given user sees a fresh news feed that contains more relevant items and fewer ignored items. For Pinot, this means that every news feed view sends several queries to Pinot to fetch the list of items that have been seen by a user. Furthermore, each news feed view and scroll event sends additional events to be indexed by Pinot so that they can be made available for subsequent queries in near realtime.

Figure 16 shows the results of adding query routing optimizations to Pinot, as described in section 4.4, and contrasts the performance with Druid as a baseline. The performance of Druid on this dataset is significantly better than with other datasets, although it does not scale as well as Pinot. In this scenario, we can see that while performance at low query rates is similar between unpartitioned and partitioned tables, adding partition awareness on the broker limits the amount of overhead as the query rate increases, leading to a significantly flatter latency curve.

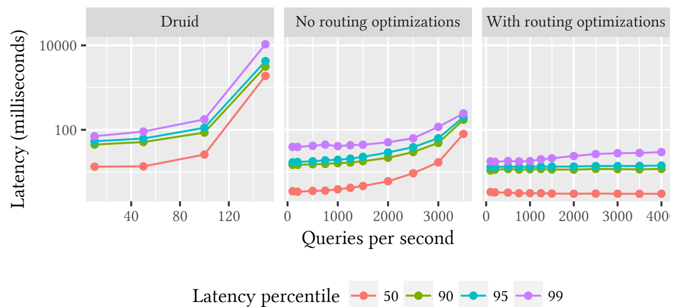

Figure 16: Comparison of routing optimizations on the impression discounting dataset

7 CONCLUSION

We have described the architecture of a production-grade system that is used to serve tens of thousands of queries per second in a demanding web environment, as well as some of the lessons learned from scaling such a system. We have also shown that a single system can process a wide range of commonly encountered analytical queries from a large web site. Finally, we have also compared how Druid, a system similar to Pinot, performs with production data and queries from a large professional network, as well as the impact of various indexing techniques and optimizations implemented in Pinot.

Future work includes adding additional types of indexes and specialized data structures for query optimization and observing their effects on query performance and service scalability.

REFERENCES

[1] 2016. Presto: Distributed SQL Query Engine for Big Data. (2016). https://prestodb.io/

[2] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 671–682.

[3] Kevin Beyer and Raghu Ramakrishnan. 1999. Bottom-up computation of sparse and iceberg cube. In *ACM SIGMOD Record*, Vol. 28. ACM, 359–370.

[4] MKABV Bittorf, Taras Bobrovytsky, Casey Ching Alan Choi Justin Erickson, Martin Grund Daniel Hecht, Matthew Jacobs Ishaan Joshi Lenni Kuff, Dileep Kumar Alex Leblang, Nong Li Ippokratis Pandis Henry Robinson, David Rorke Silvius Rus, John Russell Dimitris Tsirogiannis Skye Wanderman, and Milne Michael Yoder. 2015. Impala: A modern, open-source SQL engine for Hadoop. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research*.

[5] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *CIDR*, Vol. 5. 225–237.

[6] Samy Chambi, Daniel Lemire, Robert Godin, Kamel Boukhalfa, Charles R Allen, and Fangjin Yang. 2016. Optimizing druid with roaring bitmaps. In *Proceedings of the 20th International Database Engineering & Applications Symposium*. ACM, 77–86.

[7] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. 2016. Better bitmap performance with roaring bitmaps. *Software: practice and experience* 46, 5 (2016), 709–719.

[8] C. Chen. 2005. Top 10 unsolved information visualization problems. *Computer Graphics and Applications, IEEE* 25, 4 (july-aug. 2005), 12 – 16. https://doi.org/10.1109/MCG.2005.91

[9] Jack Chen, Samir Jindel, Robert Walzer, Rajkumar Sen, Nika Jimsheleishvilli, and Michael Andrews. 2016. The MemSQL Query Optimizer: A modern optimizer for real-time analytics in a distributed database. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1401–1412.

[10] Jeffrey Dean, Sanjay Ghemawat, and Google Inc. 2004. MapReduce: simplified data processing on large clusters. In *In OSDI04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. USENIX Association.

[11] Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey D Ullman. 1999. Computing Iceberg Queries Efficiently.. In *Internaational Conference on Very Large Databases (VLDB'98), New York, August 1998*. Stanford InfoLab.

[12] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2012. SAP HANA database: data management for modern business applications. *ACM Sigmod Record* 40, 4 (2012), 45–51.

[13] Kishore Gopalakrishna, Shi Lu, Zhen Zhang, Adam Silberstein, Kapil Surlaker, Ramesh Subramonian, and Bob Schulman. 2012. Untangling cluster management with Helix. In *ACM Symposium on Cloud Computing, SOCC '12, San Jose, CA, USA, October 14-17, 2012*. 19.

[14] Alexander Hall, Olaf Bachmann, Robert Büssow, Silviu Gănceanu, and Marc Nunkesser. 2012. Processing a trillion cells per mouse click. *Proceedings of the VLDB Endowment* 5, 11 (2012), 1436–1446.

[15] Jeffrey Heer and Ben Shneiderman. 2012. Interactive Dynamics for Visual Analysis. *Queue* 10, 2, Article 30 (Feb. 2012), 26 pages. https://doi.org/10.1145/2133416.2146416

[16] Jean-Francois Im, Michael J McGuffin, and Rock Leung. 2013. GPLOM: the generalized plot matrix for visualizing multidimensional multivariate data. *Visualization and Computer Graphics, IEEE Transactions on* 19, 12 (2013), 2606–2614.

[17] Jean-François Im, Félix Giguère Villegas, and Michael J. McGuffin. 2013. VisReduce: Fast and responsive incremental information visualization of large datasets. (2013). Proceedings of the IEEE Big Data Visualization Workshop 2013.

[18] Chris Johnson. 2004. Top scientific visualization research problems. *IEEE Computer Graphics and Applications (CG&A)* 24 (2004).

[19] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*. 1–7.

[20] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, et al. 2015. Oracle database in-memory: A dual format in-memory database. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 1253–1258.

[21] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The vertica analytic database: C-store 7 years later. *Proceedings of the VLDB Endowment* 5, 12 (2012), 1790–1801.

[22] Pei Lee, Laks VS Lakshmanan, Mitul Tiwari, and Sam Shah. 2014. Modeling impression discounting in large-scale recommender systems. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 1837–1846.

[23] Nathan Marz and James Warren. 2015. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co.

[24] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 330–339.

[25] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, et al. 2013. DB2 with BLU acceleration: So much more than just a column store. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1080–1091.

[26] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. 2005. C-store: a column-oriented DBMS. In *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 553–564.

[27] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1626–1629.

[28] Lili Wu, Roshan Sumbaly, Chris Riccomini, Gordon Koo, Hyung Jin Kim, Jay Kreps, and Sam Shah. 2012. Avatara: OLAP for web-scale analytics products. *Proceedings of the VLDB Endowment* 5, 12 (2012), 1874–1877.

[29] Dong Xin, Jiawei Han, Xiaolei Li, and Benjamin W Wah. 2003. Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*. VLDB Endowment, 476–487.

[30] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. 2014. Druid: A real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 157–168.

[31] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. *HotCloud* 10 (2010), 10–10.

[32] Hongwei Zhao and Xiaojun Ye. 2013. A multidimensional OLAP engine implementation in key-value database systems. In *Workshop on Big Data Benchmarks*. Springer, 155–170.