LinkedIn Data Infrastructure

Rajashekar[1] Nagarjuna[2] Balavineeth[3] Rishabh[4]

Abstract— **Enterprise applications are successful if they offer zero downtimes and high reliable services to the users around the world. Enterprise applications should be in a position to extend the services to the growing users and handle large amount of increasing traffic at some peak hours. For achieving these goals applications have to build on scalable, event-driven, low latency and resilient platforms. LinkedIn builds on various open source platforms which offer these services of scaling the data, being event driven and resilience out of the box. LinkedIn moves the data from users to the data processing systems in doing so LinkedIn uses publish/subscribe model (event driven model) which makes it scalable unlike the traditional database triggers which reside inside the database and operate on the data when data is added to the database. The most important requirements of the successful enterprise are Scalability, responsiveness, event driven services (non-blocking) and resilience. LinkedIn focuses at these goals at every stage while building its data infrastructure.**

I. INTRODUCTION

LinkedIn logically divides into three tiers of traditional services for responsibility division and better management. (1)Data tier: maintains the persistent state of the application including the user data. (2)Service tier: implements an API layer the makes the necessary API calls to the persistent store for fetching required data. (3)Display/Presentation tier: All that user sees and interacts with consists of presentation layer.

Going without state has added benefits to the service component or tier. Here going without state means pushing the state down to the persistent storage system instead of maintaining it in the service or display tier services. When there is no state in these tiers, the machines running these teirs become completely interchangeable. Load balancing can be done just my adding the more hardware. Failed machines can be removed from the hardware pools as soon as certain hardware fails without any downtime. User request can be routed to any machine and effects are indistinguishable from one another.

All these tiers communicate with each other using API calls. These tiers physically run on different hardware and are mostly stateless. Data volume requires partitioning the data across different machines and maintaining multiple copies of the data on multiple machines. This helps both in availability of the system and reliability in the failover cases. Replicated state requires reasoning about the consistency of the data across replicated systems. Expansion requires redistribution of the data across multiple systems without downtime.

LinkedIns core data systems are (1)Live Storage systems (2)Stream systems (3)search (4)Social graph (5)Recommendations

Live Storage Systems are the workhorses of the web applications serving most of the content required for the users. Live storage systems are nothing but the traditional OLTP databases tuned to serve the needs of the LinkedIn web application. Complex query capability of the systems is sacrificed for the sake of horizontal scalability. The two main storage systems at LinkedIn are (1)Espresso: datastore with rich query capabilities and timeline consistency model (2)Voldemort: key value, available, low latency datastore comes with tunable consistency. Inspired from Amazon dynamo and works like distributed hash table. Batch processing consists of large-scale offline data processing jobs that run on a fixed schedule, say, hourly, daily, or weekly. Much of the most complex algorithmic data processing work is offloaded to this setting.

Recommendation systems and social search systems are both closely related. These systems are concerned with matching rich structured data to users in a highly personalized way. The recommendation system matches relevant jobs, job candidates, connections, ads, news articles, and other content to users.

The search system powers people search, which is a core feature for LinkedIn, but also handles search over all other LinkedIn content (including verticals for jobs, groups, and companies).

Stream systems provide feeds of data to applications and many of the other data systems mentioned above. These feeds can be used either for processing or notification purposes.

LinkedIns two stream systems: Databus which handles database stream replication and Kafka, which handles general pub/sub, user activity events and log data.

II. VOLDEMORT

Project Voldemort is a highly available, low-latency distributed data store. It was initially developed at LinkedIn in 2008 in order to provide key-value based storage for read-write data products like Who viewed my profile, thereby acting an alternative to primary storage Oracle system.

A. Voldemort features

- Data is automatically replicated over multiple servers
- Data is automatically partitioned so each server contains only a subset of the total data
- Provides tunable consistency (strict quorum or eventual consistency)
- Server failure is handled transparently
- Pluggable Storage Engines – BDB-JE, MySQL, Read-Only

- Pluggable serialization – Protocol Buffers, Thrift, Avro and Java Serialization
- Data items are versioned to maximize data integrity in failure scenarios without compromising availability of the system
- Each node is independent of other nodes with no central point of failure or coordination
- Good single node performance: you can expect 10-20k operations per second depending on the machines, the network, the disk system, and the data replication factor
- Support for pluggable data placement strategies to support things like distribution across data centers that are geographically far apart.

B. Comparison to relational databases

- Voldemort combines in memory caching with the storage system so that a separate caching tier is not required (instead the storage system itself is just fast)
- Unlike MySQL replication, both reads and writes scale horizontally
- Data portioning is transparent, and allows for cluster expansion without rebalancing all data
- Data replication and placement is decided by a simple API to be able to accommodate a wide range of application specific strategies
- The storage layer is completely mockable so development and unit testing can be done against a throw-away in-memory storage system without needing a real cluster (or even a real storage system) for simple testing.

III. DATABUS

A. Introduction

Databus is a framework which delivers the database changes to the application running into the downstream. Now a days most of the Internet based systems are facing difficulty in performing complex processing over large data within limited time. LinkedIn also maintains the indexes and statistical models of the of the on-line identity of constantly increasing user base of 135+ million users. Databus is a system for change data capture (CDC) . CDC is the software design pattern used to determine the data that has changed so that action can be taken using the changed data. It also provides a common pipeline for transporting CDC events from LinkedIn primary databases to various applications. These applications includes the people search index, Read Replicas,Social Graph Index.

B. Requirements

Here we will discuss the requirement for the databus .

- Strong timeline consistency: It is highly required to avoid having the subscriber see partial data.
- User-space processing: This is ability to perform the computation triggered by the data change outside the database server. It reduces the load of the database server as the computation will be done into the user-space and it also enables independent scaling of the subscribers.
- Support for long look-back queries: It provides the support to the subscriber to read the older changes when the subscriber want to change the things that have not processed yet. It also provides the recent snapshot of the database when a new subscriber want to initialize their state to increase the capacity. Sometimes all client need to re-initialize their state because of the need to reprocess the whole data set, like processing algorithm changes.
- Data source / subscriber isolation: It isolates the source database from the number of subscribers so that increasing the number of the latter should not impact the performance of the former. It isolates the source database from slow or failing subscribers that should not negatively impact the database performance.

Databus guarantees the following :

1) Timeline consistency
2) No loss in durability
3) High availability
4) Low latency

It guarantees that there will be no loss in durability because it relies on the bootstrap server and the primary store on the failure . It guarantees that their will be high availability because it replicates the data between different relays and bootstrap server .As there will be 100 s of relays to processing their will be no delay . Hence the latency in processing is very small i.e in the order of milliseconds. The above guarantees can be better understood by the architecture of the data bus so in next topic we are going to discuss about its architecture.

C. Architecture

The Data Bus consists of three main components: the Relay, the Bootstrap Server, and the Databus Client Library. The Relay captures changes in the source database, serializes them to a common binary format and buffers those. Change is represented by a Databus CDC event which contains a sequence number in the commit order of the source database, metadata, and payload with the serialized change. Relays serve Databus events to both clients and the Bootstrap servers. Bootstrap servers provide clients with arbitrary long look-back queries in Data Bus event stream and isolate the source database from having to handle these queries.

Now, we will describe the three main components in detail.

- RELAY: The very first task of the relay is to capture changes from the source database. LinkedIn uses triggers or consuming from the database replication log to capture . Once captured , the changes are serialized to a data source independent binary format. Linked uses Avro for binding the multiple languages. The serialized events are stored in a circular in memory buffer and then databus client are served using the circular in memory buffer.

  Relay provides the following :

  1) Efficient buffering of tens of GB of data with hundreds of millions of databus events.

2) It Supports the hundreds of consumers per relay without the contribution of the source database.
3) It servers with very low latency.
4) Multiply partitioning schemes to the client will be served by the server-side filtering .

- Bootstrap Server: It listen to the stream of the database event and provide long term storage for them .Due to the limited memory sometime relay will not be able to serve to the client so it will serve the same request to decrease the latency period. Thus it isolates the primary database from having to serve to those client.
- Databus client library: It is the business logic of the database consumers and it is also the connection between the Relays and the Bootstrap server.It provides the following :
  1) It retry logic if the consumers fails to process any event .
  2) Multi-thread processing will be supported.
  3) Continuous stream of the databus events will be ensured by the local buffering and the flow control.
  4) Push and pull interface.

IV. Espresso

A. Introduction

Espresso is a horizontally scalable, indexed, timeline-consistent, document-oriented, highly available NoSQL data store. With the increase in the data and increased number of users their requirements are exceeding the capabilities of traditional RDBMS system. In the earlier days, LinkedIn was very simple. It used the traditional RDBMS for handling the tasks on handful of tables for users, groups, companies or organizations etc. Over the years as Data increased LinkedIn also evolved. The early pattern of primary, strongly consistent, data store which accepts the reads and writes, then changes the values in the table have become common design pattern. The Primary data requirements of LinkedIn did not require the full functionality of a RDBMS. In fact using RDBMS was not that useful and has some disadvantages like, the existing RDBMS installation is very costly which requires costly, specialized hardware and extensive caching to meet the requirements of the latency. Second it was difficult to do scale with 100percent uptime.

B. Why Espresso is Needed

In 2009, LinkedIN has introduced Voldemort to the data ecosystem. Voldemort was a key:value type of data storage which is simple, eventually consistent. In 2011, LinkedIn identified that there were several key points that needed to be covered. The points are:

- Scale and Elasticity
- Consistency
- Integration
- Bulk Operations
- Need of Secondary Indexing
- Schema Independent

To meet all the needs or requirements, they have introduced a data model design called Espresso.

C. Features

- Transaction support: We all know that most of the NoSQL stores do not support transaction beyond a single record/document. A large number of use-cases at LinkedIn brings them to natural partitioning of the data into collections that share a common partitioning key. In most cases partitioning key generally will be the primary key. Example like memberid, userid, groupid etc.
- Consistency Model: CAP theorem states that a distributed system can only achieve any two out of consistency, availability and partition-failure tolerance.
- Integration with complete data ecosystem: Providing a data platform as a service for application developers is a big advantage and espresso exactly does this. It's another goal is to support developer agility by ensuring tight integration with with the rest of the data ecosystem at LinkedIn.
- Schema Awareness and Rich Functionality: Many other NoSQL stores that are schema-free, Espresso supports schema definition. Espresso supports or compatible with many schema. It also supports schema definition for the documents. Espresso avoids the rigidity of the traditional RDBMS by allowing on-the-fly Schema evolution. This on-the-fly schema, we can add any new field to the document at any time and this change is backward compatible across the entire ecosystem.

D. Data Model

Expresso's data model has been developed from the observations and use-cases and patterns at LinkedIn. LinkedIn wanted some thing which is in between the simple Key:Value storage and RDBMS. They wanted a Hybrid model which is scalable, timeline consistent and provides secondary indexing. Espresso uses a hierarchical data model. The data hierarchy composed of Databases, Documents Groups, Tables and last Documents. Below is the description of each of the following data models.

- Document: It is the smallest unit of data representation in Espresso. They are logically hashed to entities, to data structures like lists, arrays and maps as fields within the document. In any RDBMS model, they are like a row in the table which are identified by a primary key.
- Table: An espresso table is a collection of like-schemed-documents. It is similar to relational database, in which table defines a unique key for each and every document that it stores. Espresso in addition to that also supports another feature called auto-generation of keys.
- Document-Group: As the name indicates it is group of documents that are present in the same database and share common partitioning key. These document-groups can span across the tables and form the largest unit of data transaction.
- Database: It is the largest storage unit of Espresso. It is analogous to that of the SQL in which the database is made up of tables with in them. All the

documents present with in the database are partitioned using the same partitioning technique. There are many partitioning techniques like Hash partitioning, Range partitioning etc.

E. API

Espresso offers a REST API for the ease of integration. Here we will cover the details of how the API works and uses of this API.

- Read Operations: Espresso provides document look ups using primary keys or secondary indexes. There are two ways to lookup using primary keys they are:
  1) If we specify the key then we are returned with the complete document.
  2) If we specify list of list of keys sharing the leading keys, it returns all the related documents
  3) If we specify the projection of fields of documents, the only the required fields are returned.
  4) The secondary indexes are used to perform search beyond the document group, the table on which the query needs to be run.
- Write Operations: Espresso has many insert or write operations. It can support a full update of a single document using a complete key. Many operations like partial update using the complete key, autoincrement of the partial-key and transactional update to the document groups.
- Conditionals: These are used in both the reads and writes of the documents. They currently support the simple predicates on time-last-modified and etag. It is very rare to see a conditional writes as the rich API allows fairly complex server side processing.
- Multi Operations: It is similar to a bash operation. As the name suggest, all the read and write operations grouped into one transaction.
- Change Stream Listener: This API comes from the Databus. This works by allowing an external observer to see all the changes or mutations taking place on database while preserving the commit order of the transactions within the database group. This API also captures the following information like type of change(insert, delete, update etc.), the key of the modified document, the pre and post image of the data and the SCN of the change. This allows observer to consume or use the data stream and while noticing the transaction boundaries to maintain the consistency at all the times.

F. System Architecture

- System Overview: The Espresso system consists of four major components: clients and routers, storage nodes, databus relays and cluster managers. Espresso clients generate reads and writes and these are taken to routers, routers send these requests to the system nodes which own the data. These storage nodes after getting requests from routers use primary keys or secondary index lookup. Changes to the databus are replicated from node to data bus relays and consumed by another system node in order to maintain the consistency. Cluster Manager manages and maintains the system nodes and Data nodes. It is called as Helix.
- System Components: In this section we will be discussing about all the parts in detail of the components. The description of the components is given below:
- Client and Router: An application sends a request to an Espresso endpoint by sending an HTTP request to a router, this router inspects the URI request from the client and forwards this request to the appropriate system node. The routing of requests uses a routing logic which uses partitioning method for the database schema and applies the appropriate key of URI for partition. Then the router uses a routing table which maps each partition to the master-storage node, and sends the request to the node.
- Storage Node: It is the building block for the horizontal scale. Data is partitioned and stored on the storage-nodes, which maintain base data and corresponding local secondary indexes. Storage nodes also provide local transaction with the help of a common root key. These nodes in order to maintain replicas using a change log stream provided by the replication tier. Slave partitions are updated by the change logs present in replication tier and applying them transactionally to both base data and the local secondary index. Storage-nodes in addition to serving the requests from clients and replication-tier they also run utility tasks periodically, including consistency check between the master and slaves.
- Cluster Manager: Espresso's cluster manager uses Apache Helix. This cluster manager using the cluster state and the input system constraints and resources it computes the Ideal state. It also monitors the cluster health, and redistributes resources upon node failure. Each of the Espresso database is horizontally sharded into a number of partitions as specified in the document schema and each partition having a configurable number of replicas. Of all the replicas one is confined as the Master and all the other replicas as Slaves.

V. KAFKA

A Scalable and Efficient messaging system for collecting various user activity events and log data. As LinkedIn has large amount of event data generated by the user activities like logins, page-views, clicks, sharing, comments, and search queries and other user activities like operational metrics and a component of off line analysis for tracking user engagement.Kafka adopts a API for messages which supports in both real time and offline delivering of this data. Data generated is of 2-3 orders in magnitude larger than traditional messaging systems, but designed to make system simple, efficient and scalable.

A. Kafka API and Architecture

Basic terms used in Kafka are, A stream of messages of a particular type is defined by a topic. Producer publishes messages to topic, published messages are stored at a set

of servers called brokers. Consumer can subscribe to one or more topics, and consumes subscribed messages by pulling data from the brokers. Kafka introduced iterator interface for stream of messages as if consumers has no more messages to consume, iterator blocks until the process restarts, but it will not leads to termination. So, at a high level, producers send messages over the network to the Kafka cluster which in turn serves them up to consumers like this:

B. Storage

As there are huge number of partitions, as each partition belongs to a logical log. Producer publishes messages to partition, the Broker adds the message to the last segment file. To get the better performance, segment files are being added to disk only after a configurable number of messages have been published or a certain amount of time. In kafka, messages doesn't have an explicit message id, instead it is addressed by its logical offset in log. If customer requires to compute the message ids, we have to add the length of current message to its id. Unlike traditional messaging systems supports out of order delivering of messages, leads to increase in complexity. So, kafka sends messages to consumers asynchronously by pull requests to the brokers, brokers locates the requested file by searching the offset list and sends back to consumers. Kafka has a more efficient storage format, only fewer bytes were transferred from the broker to the consumer in Kafka. On average, each message had an overhead of 9 bytes in Kafka, versus 144 bytes in ActiveMQ. As broker in both ActiveMQ and RabbitMQ had to maintain the delivery state of every message..There were no disk write activities on the Kafka broker. Kafka reduces transmission overhead by using API of send file.

C. Kafka Broker

Kafka Broker is stateless, that consumer has to maintain the amount of data they consumed. It is done by deleting message from the broker as broker doesn't know whether consumer consumed the message or not. Kafka innovatively solves this problem by using a simple time-based SLA for the retention policy. A message is automatically deleted if it has been retained in the broker longer than a certain period. This design has a benefit that consumer can get messages from an old offset and re-consume data.

D. Eficient Transfer

Kafka supports compression while transferring the messages. As producer compresses the messages and send it to the broker. In turn the compressed data is stored in broker and eventually delivered to consumer and it is uncompressed. By sending the data this way, it can save 2/3rd of the network bandwidth. kafka doesn't have any source for caching messages, but it has file system page cache which is used for buffering. As,it avoids double buffering and retains cache whenever required ie., as broker process gets restarted. While transferring Kafka producer doesnt wait for acknowledgements from the broker but sends messages as faster as the broker can handle.

E. Distributed Coordination

Consumer groups are present in Kafka, each consumer group consists of one or more consumers that consume a set of subscribed topics. From these each message is delivered to only one of the consumers in the group. Different consumer groups each independently consume messages and no coordination is required. Consumers within th same group can be in different machines or processes. If consumers from different processes or groups requires messages from one topic, then it requires coordination for overview of the process. So, to facilitate coordination, Kafka uses ZooKeeper . It is used for following tasks:

1) detecting the addition and the removal of brokers and consumers,
2) triggering a re balance process in each consumer when the above events happen, and
3) maintaining the consumption relationship and keeping track of the consumed offset of each partition.

F. Kafka Deployment at LinkedIn

LinkedIn started using Kafka since Aug. 2010. They have one Kafka cluster located with each data center, where user services are running. Services of user event data are generated at front end, and even online consumers within the same data center. They also have Kafka cluster in separate data center for off line analysis, located close to Hadoop cluster, where they run various analytical process and reporting jobs on the data. They also use this cluster for running scripts against events and for prototyping. LinkedIn, at present collects huge amount of compressed activities at a peak rate of 50k messages per second and service call events at a peak rate of 200k messages per second. For reduction of data loss, they generated time stamp for each message carrier. Kafka has been an Apache incubator project [KA] since July 2011. Now, LinkedIn wants to add one of the major feature to Kafka is intra-cluster replication.

References are important to the reader; therefore, each citation must be complete and correct. If at all possible, references should be commonly available publications.

REFERENCES

[1] Aditya Auradkar, Chavdar Botev, Shirshanka Das, Dave DeMaagd, Alex Feinberg, Phanindra Ganti, Lei Gao, Bhaskar Ghosh, Kishore Gopalakrishna, Brendan Harris, Joel Koshy, Kevin Krawez, Jay Kreps, Shi Lu, Sunil Nagaraj, Neha Narkhede, Sasha Pachev, Igor Perisic, Lin Qiao, Tom Quiggle, Jun Rao, Bob Schulman, Abraham Sebastian, Oliver Seeliger, Adam Silberstein, Boris Shkolnik, Chinmay Soman, Roshan Sumbaly, Kapil Surlaker, Sajid Topiwala, Cuong Tran, Balaji Varadarajan, Jemiah Westerman, Zach White, David Zhang, and Jason Zhang , LinkedIn Data Infrastructure, in 2012 IEEE 28th International Conference on Data Engineering

[2] Voldemort: http://www.project-voldemort.com/voldemort/

[3] Voldemort: http://en.wikipedia.org/wiki/Voldemort(distributed.data.store)

[4] Kafka: http://kafka.apache.org/