# A Brief History of Scaling LinkedIn

## [https://content.linkedin.com/content/dam/engineering/en-us/blog/migrated/josh_0_0.jpgJosh Clemm](https://engineering.linkedin.com/blog/authors/j/josh-clemm)

LinkedIn [started in 2003](https://ourstory.linkedin.com/) with the goal of connecting to your network for better job opportunities. It had only 2,700 members the first week. Fast forward many years, and LinkedIn’s product portfolio, member base, and server load has grown tremendously.

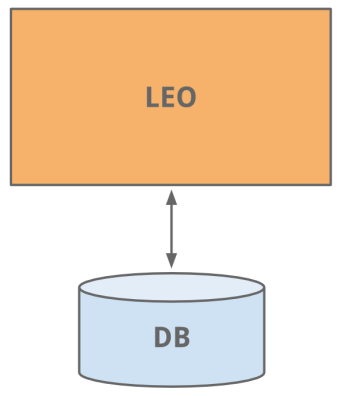
Today, LinkedIn operates globally with more than 350 million members. We serve tens of thousands of web pages every second of every day. We've hit our [mobile moment](http://blog.linkedin.com/2014/04/18/the-next-three-billion/) where mobile accounts for more than 50 percent of all global traffic. All those requests are fetching data from our backend systems, which in turn handle millions of queries per second.

***So, how did we get there?***

## The early years

### Leo

LinkedIn started as many sites start today, as a single monolithic application doing it all. That single app was called Leo. It hosted web servlets for all the various pages, handled business logic, and connected to a handful of LinkedIn databases.



Ah, the good old days of website development - nice and simple

### Member Graph

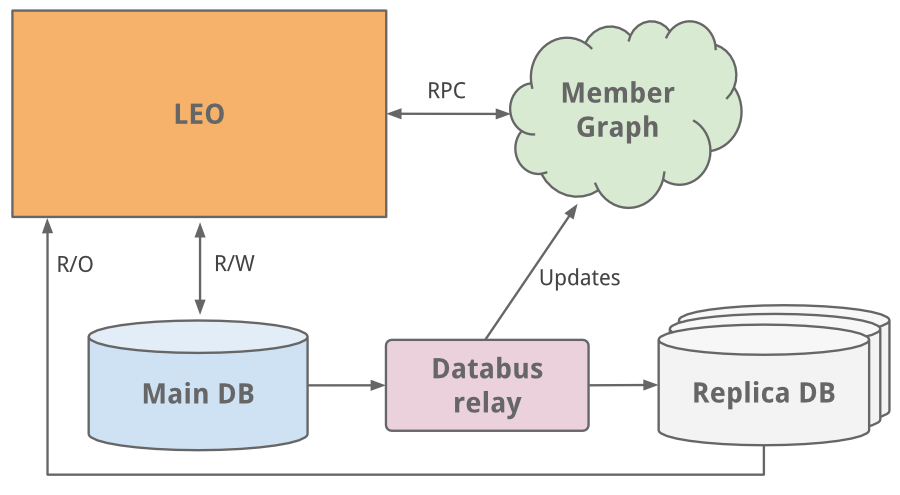
One of the first things to do as a social network is to manage member to member connections. We needed a system that queried connection data using graph traversals and lived in-memory for top efficiency and performance. With this different usage profile, it was clear it needed to scale independently of Leo, so a separate system for our **member graph** called Cloud was born - LinkedIn’s first service. To keep this graph service separate from Leo, we used Java RPC for communication.

It was around this time we needed search capabilities. Our member graph service started feeding data into a new search service running [Lucene](https://lucene.apache.org/).

### Replica read DBs

As the site grew, so did Leo, increasing its role and responsibility, and naturally increasing its complexity. Load balancing helped as multiple instances of Leo were spun up. But the added load was taxing LinkedIn’s most critical system - its **member profile database**.

An easy fix we did was classic vertical scaling - throw more CPUs and memory at it! While that bought some time, we needed to scale further. The profile database handled both read and write traffic, and so in order to scale, replica slave DBs were introduced. The replica DBs were a copy of the member database, staying in sync using the earliest version of [databus](http://data.linkedin.com/blog/2012/10/driving-the-databus) (now [open-sourced](https://engineering.linkedin.com/data-replication/open-sourcing-databus-linkedins-low-latency-change-data-capture-system)). They were set up to handle all read traffic and logic was built to know when it was safe (consistent) to read from a replica versus the main master DB.



\* While the master-slave model worked as a medium-term solution, we’ve since moved to partitioned DBs

As the site began to see more and more traffic, our single monolithic app Leo was often going down in production, it was difficult to troubleshoot and recover, and difficult to release new code. High availability is critical to LinkedIn. It was clear we needed to “Kill Leo” and break it up into many small functional and [stateless services](http://en.wikipedia.org/wiki/Service-oriented_architecture).

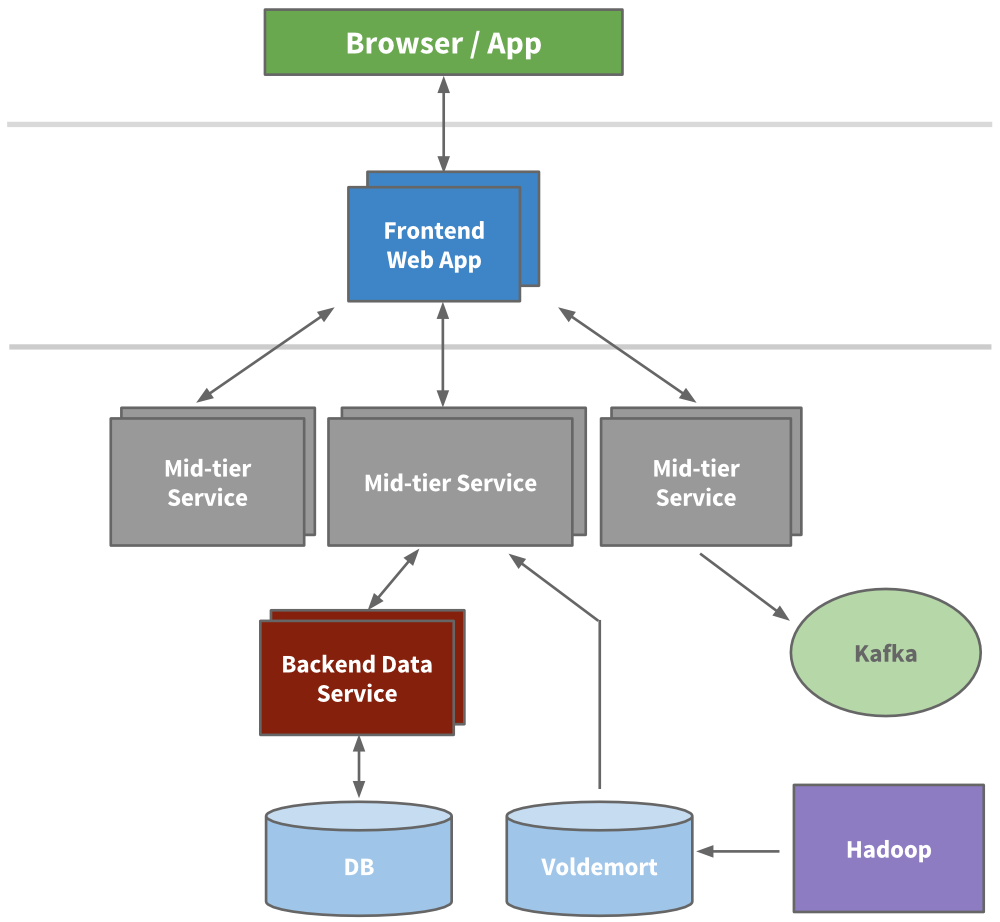


“Kill Leo” was the mantra internally for many years...

### Service Oriented Architecture

Engineering started to extract micro services to hold APIs and business logic like our search, profile, communications, and groups platforms. Later, our presentation layers were extracted for areas like our recruiter product or public profile. For new products, brand new services were created outside of Leo. Over time, vertical stacks emerged for each functional area.

We built frontend servers to fetch data models from different domains, handle presentation logic, and build the HTML (via JSPs). We built mid-tier services to provide API access to data models and backend data services to provide consistent access to its database(s). By 2010, we already had over 150 separate services. Today, we have over 750 services.



An example multi-tier service oriented architecture within LinkedIn

Being stateless, scaling could be achieved by spinning up new instances of any of the services and using hardware load balancers between them. We actively started to redline each service to know how much load it could take, and built out early provisioning and performance monitoring capabilities.

### Caching

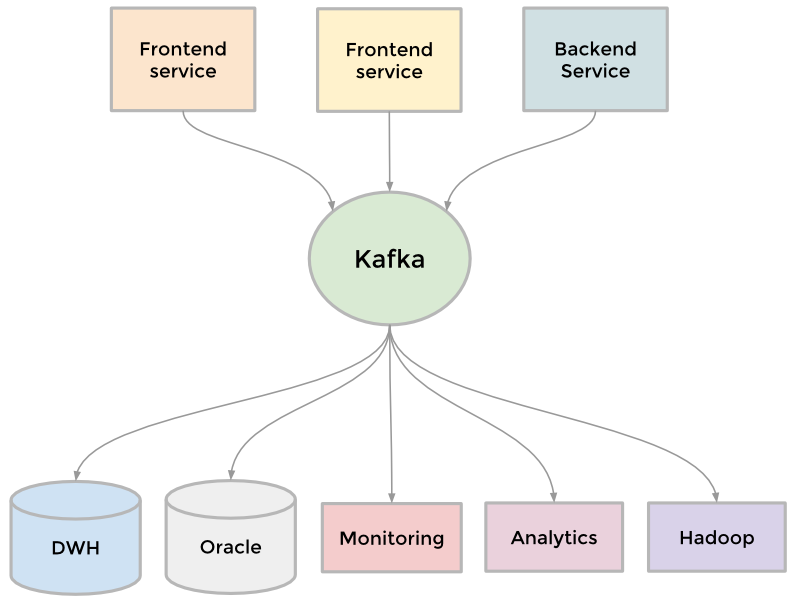
LinkedIn was seeing hypergrowth and needed to scale further. We knew we could reduce the load altogether by adding more layers of cache. Many applications started to introduce mid-tier caching layers like [memcache](https://en.wikipedia.org/wiki/Memcached) or [couchbase](https://en.wikipedia.org/wiki/Couchbase_Server). We also added caches to our data layers and started to use [Voldemort](http://engineering.linkedin.com/tags/voldemort) with precomputed results when appropriate.

Over time, we actually removed many mid-tier caches. Mid-tier caches were storing derived data from multiple domains. While caches appear to be a simple way to reduce load at first, the complexity around invalidation and the call graph was getting out of hand. Keeping the cache closest to the data store as possible keeps latencies low, allows us to scale horizontally, and reduces the cognitive load.

### Kafka

To collect its growing amount of data, LinkedIn developed many custom data pipelines for streaming and queueing data. For example, we needed our data to flow into data warehouse, we needed to send batches of data into our [Hadoop workflow](http://data.linkedin.com/projects/hadoop) for analytics, we collected and aggregated logs from every service, we collected tracking events like pageviews, we needed queueing for our inMail messaging system, and we needed to keep our people search system up to date whenever someone updated their profile.

As the site grew, more of these custom pipelines emerged. As the site needed to scale, each individual pipeline needed to scale. Something had to give. The result was the development of [Kafka](http://blog.confluent.io/2015/02/25/stream-data-platform-1/), our distributed pub-sub messaging platform. Kafka became a universal pipeline, built around the concept of a [commit log](http://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying), and was built with speed and scalability in mind. It enabled near realtime access to any data source, empowered our Hadoop jobs, allowed us to build [realtime analytics](http://engineering.linkedin.com/analytics/real-time-analytics-massive-scale-pinot), vastly improved our [site monitoring](http://engineering.linkedin.com/samza/real-time-insights-linkedins-performance-using-apache-samza) and [alerting capability](http://engineering.linkedin.com/52/autometrics-self-service-metrics-collection), and enabled us to visualize and [track our call graphs](http://engineering.linkedin.com/distributed-service-call-graph/real-time-distributed-tracing-website-performance-and-efficiency). Today, Kafka handles well over [500 billion events per day](http://engineering.linkedin.com/kafka/kafka-linkedin-current-and-future).



Kafka as the universal data stream broker

### Inversion

Scaling can be measured across many dimensions, including organizational. In late 2011, LinkedIn kicked off an internal initiative called [Inversion](http://www.bloomberg.com/bw/articles/2013-04-10/inside-operation-inversion-the-code-freeze-that-saved-linkedin). This initiative put a pause on feature development and allowed the entire engineering organization to focus on improving tooling and deployment, infrastructure, and developer productivity. It was successful in enabling the engineering agility we need to build the scalable new products we have today.

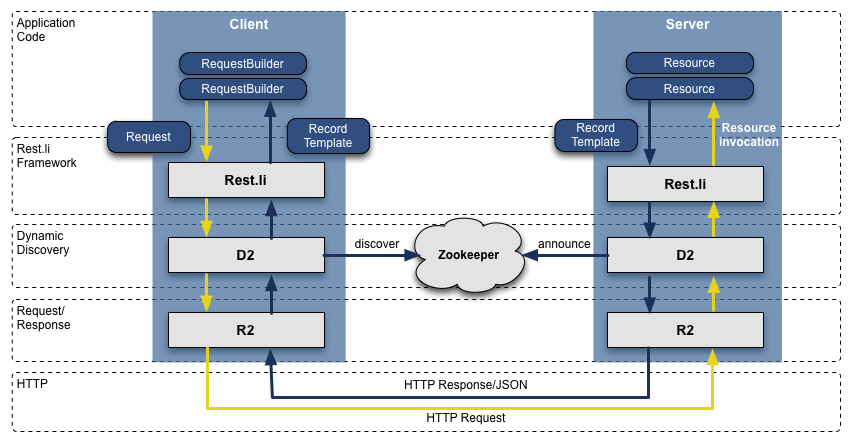
## The modern years

### Rest.li

When we transformed from Leo to a service oriented architecture, the APIs we extracted assumed Java-based RPC, were inconsistent across teams, were tightly coupled with the presentation layer, and it was only getting worse. To address this, we built out a new API model called [Rest.li](http://engineering.linkedin.com/architecture/restli-restful-service-architecture-scale). Rest.li was our move towards a data model centric architecture, which ensured a consistent stateless Restful API model [across the company](http://engineering.linkedin.com/restli/linkedins-restli-moment).

By using JSON over HTTP, our new APIs finally made it easy to have non-Java-based clients. LinkedIn today is still mainly a Java shop, but also has many clients utilizing Python, Ruby, Node.js, and C++ both developed in house as well as from tech stacks of our acquisitions. Moving away from RPC also freed us from high coupling with presentation tiers and many backwards compatibility problems. Plus, by using [Dynamic Discovery (D2)](https://github.com/linkedin/rest.li/wiki/Dynamic-Discovery) with Rest.li, we got automated client based load balancing, discovery, and scalability of each service API.

Today, LinkedIn has over 975 Rest.li resources and over 100 billion Rest.li calls per day across all our datacenters.



Rest.li R2/D2 tech stack

### Super Blocks

Service oriented architectures work well to decouple domains and scale services independently. But there are downsides. Many of our applications fetch many types of different data, in turn making hundreds of downstream calls. This is typically referred to as a “call graph”, or “fanout” when considering all the many downstream calls. For example, any Profile page request fetches much more beyond just profile data including photos, connections, groups, subscription info, following info, long form blog posts, connection degrees from our graph, recommendations, etc. This call graph can be difficult to manage and was only getting more and more unruly.

We introduced the concept of a super block - groupings of backend services with a single access API. This allows us to have a specific team optimize the block, while keeping our call graph in check for each client.

### Multi-Data Center

Being a global company with a fast growing member population, we needed to scale beyond serving traffic [from one data center](https://www.linkedin.com/pulse/armen-hamstra-how-he-broke-linkedin-got-promoted-angel-au-yeung). We began an effort years ago to address this, first by serving public profiles out of two data centers (Los Angeles and Chicago). Once proven, we embarked on enhancing all our services to handle data replication, callbacks from different origins, one-way data replication events, and pinning users to a geographically close data center.

Many of our databases run on [Espresso](http://engineering.linkedin.com/espresso/introducing-espresso-linkedins-hot-new-distributed-document-store) (a new in-house multi-tenant datastore). Espresso was built with multi data centers in mind. It provides master / master support and handles much of the difficult replication.

Multiple data centers are incredibly important to maintain “site-up” and high availability. You need to avoid any single point of failure not just for each individual service, but the entire site. Today, LinkedIn runs out of three main data centers, with additional [PoPs](http://engineering.linkedin.com/performance/how-linkedin-used-pops-and-rum-make-dynamic-content-download-25-faster) around the globe.



LinkedIn's operational setup as of 2015 (circles represent data centers, diamonds represent PoPs)

### What else have we done?

Of course, our scaling story is never this simple. There’s a countless number of things we’ve done over the years across all engineering and operations teams, including some of these larger initiatives:

Many of our most critical systems have their own rich history and evolution to address scale over the years. This includes our [member graph service](http://engineering.linkedin.com/real-time-distributed-graph/using-set-cover-algorithm-optimize-query-latency-large-scale-distributed) (our first service outside from Leo), [search](https://engineering.linkedin.com/search/did-you-mean-galene) (our second service), news feed, [communications platform](http://www.slideshare.net/manivannan57/LinkedIn-Communication-Architecture-Presentation-2?related=5), and member profile backend.

We’ve built data infrastructure that enables long term growth. This was first evident with Databus and Kafka, and has continued with [Samza](https://engineering.linkedin.com/samza/real-time-insights-linkedins-performance-using-apache-samza) for data streams, [Espresso](http://engineering.linkedin.com/espresso/introducing-espresso-linkedins-hot-new-distributed-document-store) and Voldemort for storage solutions, [Pinot](http://engineering.linkedin.com/analytics/real-time-analytics-massive-scale-pinot) for our analytics systems, as well as other custom solutions. Plus, our tooling has improved such that developers can [provision this infra automatically](https://www.linkedin.com/pulse/invisible-infrastructure-alex-vauthey).

We’ve developed a massive offline workflow using [Hadoop](http://data.linkedin.com/projects/hadoop) and our [Voldemort data store](http://engineering.linkedin.com/tags/voldemort) to precompute data insights like People You May Know, Similar profiles, Notable Alumni, and profile browse maps.

We’ve rethought our frontend approach, adding [client templates](http://www.slideshare.net/brikis98/dustjs) into the mix ([Profile page](http://engineering.linkedin.com/profile/engineering-new-linkedin-profile), [University pages](http://engineering.linkedin.com/university/building-linkedin-university-pages)). This enables more interactive applications, requiring our servers to send only JSON or partial JSON. Plus, templates get cached in CDNs and the browser. We also started to use [BigPipe](http://engineering.linkedin.com/frontend/new-technologies-new-linkedin-home-page) and the [Play framework](https://engineering.linkedin.com/play/composable-and-streamable-play-apps), changing our model from a threaded web server to a [non-blocking asynchronous](https://engineering.linkedin.com/play/play-framework-async-io-without-thread-pool-and-callback-hell) one.

Beyond the application code, we’ve introduced [multiple tiers of proxies](http://www.slideshare.net/thenickberry/reflecting-a-year-after-migrating-to-apache-traffic-server) using Apache Traffic Server and HAProxy to handle load balancing, data center pinning, security, intelligent routing, server side rendering, and more.

And finally, we continue to improve the performance of our servers with optimized hardware, [advanced memory](http://engineering.linkedin.com/garbage-collection/garbage-collection-optimization-high-throughput-and-low-latency-java-applications) and [system](http://engineering.linkedin.com/performance/optimizing-linux-memory-management-low-latency-high-throughput-databases) tuning, and utilizing newer Java runtimes.

## What’s next

LinkedIn continues to grow quickly and there’s still a ton of work we can do to improve. We’re working on problems that very few ever get to solve - [come join us!](https://www.linkedin.com/company/linkedin/careers?trk=eng-blog)

*Thanks to* [*Steve*](https://www.linkedin.com/in/stevenihde)*,* [*Swee*](https://www.linkedin.com/in/sweelim)*,* [*Venkat*](https://in.linkedin.com/in/ganesanvenkatasubramanian)*,* [*Eran*](https://www.linkedin.com/in/eranl)*,* [*Ram*](https://www.linkedin.com/in/ramvem)*,* [*Brandon*](https://www.linkedin.com/in/bchesla)*,* [*Mammad*](https://www.linkedin.com/in/mammadzand)*, and* [*Nick*](https://www.linkedin.com/in/nberry) *for the help in reviewing.*

# Using set cover algorithm to optimize query latency for a large scale distributed graph

## [https://content.linkedin.com/content/dam/engineering/en-us/blog/migrated/35264f7_0.jpgRui Wang](https://engineering.linkedin.com/blog/authors/r/rui-wang)

Social networks often require the ability to perform low latency graph computations in the user request path. For example, at LinkedIn, we show the graph distance and common connections whenever we show a profile on the site. To do this, we have developed a distributed and partitioned graph system that scales to hundreds of millions of members and their connections and handles hundreds of thousands of queries per second. We published a paper in the [HotCloud'13 Conference, June 2013](https://www.usenix.org/conference/hotcloud13) that describes one of the techniques we use to keep latencies low:

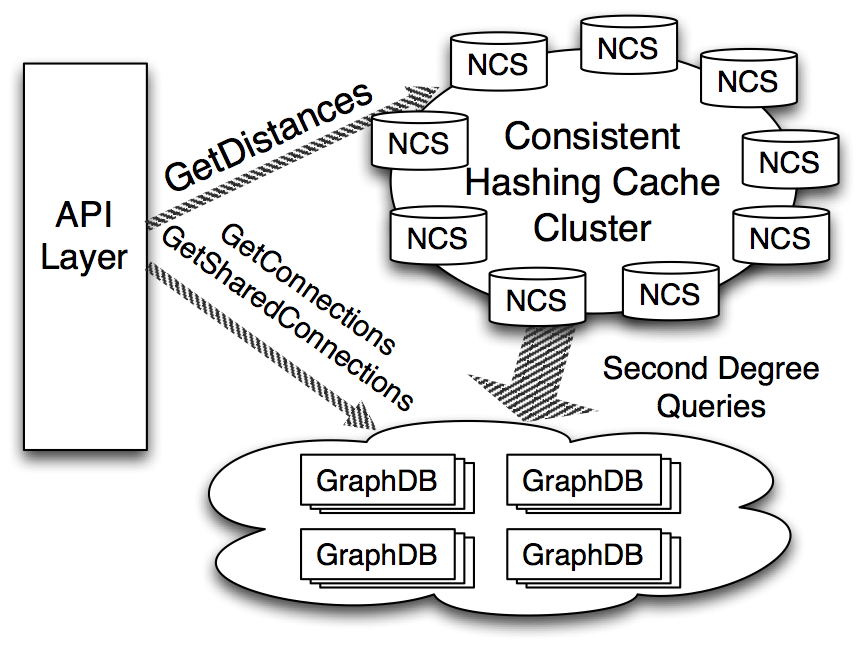
[**Using Set Cover to Optimize a Large-Scale Low Latency Distributed Graph**](http://0b4af6cdc2f0c5998459-c0245c5c937c5dedcca3f1764ecc9b2f.r43.cf2.rackcdn.com/11567-hotcloud13-wang.pdf)

In this post, I'll describe the greedy set cover algorithm we developed and how it reduces latencies for more than half the queries in our real-time distributed graph infrastructure.

## Overview of Query Routing

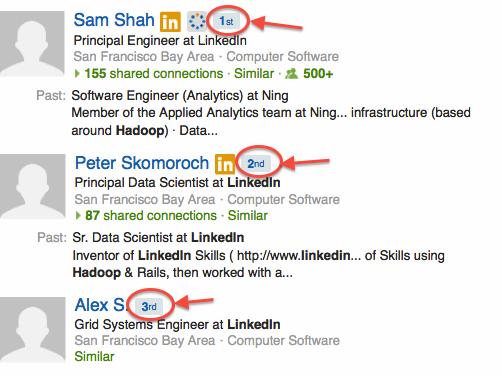
LinkedIn's distributed graph system consists of three major subcomponents:

1. **GraphDB**: a partitioned and replicated graph database.
2. **Network Cache Service (NCS)**: a distributed cache that stores a member's network and serves queries requiring second degree knowledge.
3. **API layer**: the access point for front-ends.



Graph Architecture

A key challenge in a distributed graph system is that, when the graph is partitioned across a cluster of machines, multiple remote calls must occur during graph traversal operations. At LinkedIn, more than half the graph queries are for calculating distance between a member and a list of member IDs up to three degrees. In general, anywhere you see a distance icon on the LinkedIn site, there's a graph distance query at the backend. So, scalability and latency for this call are major considerations.



Distance Icons

NCS, the caching layer, calculates and stores a member's second-degree set. With this cache, graph distance queries originating from a member can be converted to set intersections, avoiding further remote calls. For example, if we have member X's second degree calculated, to decide whether member Y is three degree apart from member X, we can simply fetch Y's connections and intersect X's second degree cache with Y's first degree connections.

A member's second-degree set is built in real time on every visit. On average, this cache has a hit ratio of over 90% to serve graph traversal queries. When cache miss occurs, the network cache service builds the cache in real time by gathering members' second-degree connection information from the GraphDB cluster. So the long tail of the graph distance queries comes from second-degree network cache miss. We use a brute force algorithm to compute this cache. The member's second-degree connections are gathered from several GraphDB nodes that store them. Each GraphDB node performs carefully tuned parallel merges to construct the partial result before sending the data back. A single NCS node is responsible for merging all partial results into one final second-degree array.

## The Scaling Problem

During an effort to scale second-degree-network computation, we discovered that a much larger number of GraphDB nodes were needed to return a member's second-degree connections during cache creation. For example, say that a member has two connections X and Y, with X stored on partition Px, and Y stored on partition Py. We found that even if there was a GraphDB node storing both Px and Py, the routing layer was more likely to route the query to two different GraphDB nodes for Px and Py separately, causing the merging of connections X and Y to complete on the NCS node, rather than on the GraphDB nodes.

## Set Cover Algorithm

We decided to apply a [greedy set cover algorithm](http://en.wikipedia.org/wiki/Set_cover_problem) to address this query optimization problem. Greedy set cover algorithms are used to ﬁnd the smallest subset that covers the maximum number of uncovered points in a large set. In this case, we would apply the algorithm to the set of partitions that stored a member's first-degree connections. Partitions stored on each GraphDB node would be the elements in a family of sets. We wanted to find the smallest number of elements from this set family that covered the input set.

[The classic implementation](http://en.wikipedia.org/wiki/Set_cover_problem#Greedy_algorithm) of the greedy set cover algorithm worked very well to reduce the number of GraphDB nodes requested during second-degree cache computation, but it introduced a noticeable latency during nodes discovery. This latency was caused by a large number of set intersections done in each of the greedy selection iterations.

## Enhancements to the Algorithm

We were able to modify this greedy algorithm by taking advantage of an additional property of our system: GraphDB nodes belonging to the same replica provide one copy of the entire graph, and there is no partition overlap among the nodes. We concluded that nodes from the same replica were more likely to provide greater coverage.

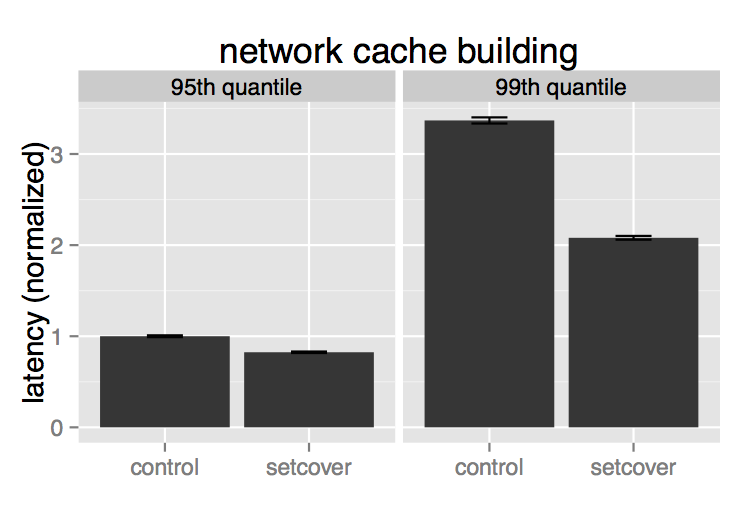
We started with a randomly selected partition from the set to be covered, and performed greedy set intersections only across the replicas covering the randomly picked partition and the entire set. By doing this, we had two guarantees:

* During each iteration, the algorithm would select two nodes that belongs to the same replica, or a node that offers equivalent coverage, or a node that removes at least the randomly picked partition.
* If we always checked at least one GraphDB node from each replica, we had a good chance of locating a node that provided close to optimal coverage, without having to examine every single GraphDB node in the cluster. This is because nodes belonging to the same replica as the already selected nodes tend to offer greater partition coverage.

## Evaluation

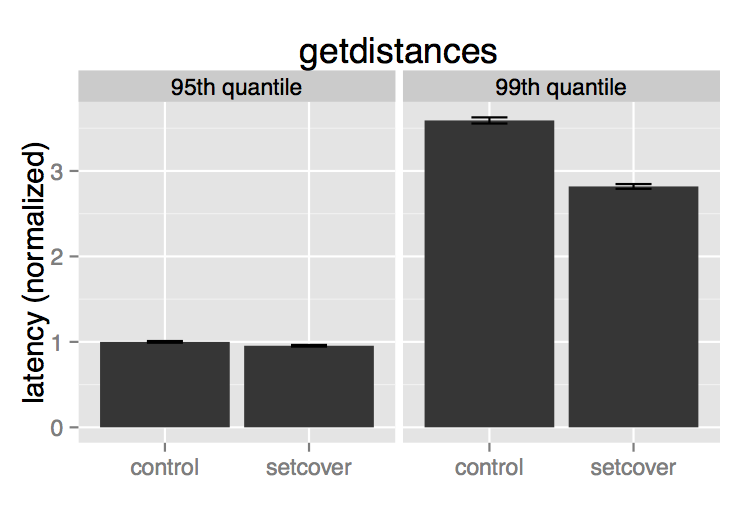
We were able to apply this set cover algorithm in our production environment without introducing additional routing latency.

Second-degree cache creation time dropped by 38% in the 99th percentile.



Comparison of 99th-percentile second-degree cache creation time, control vs setcover

We also saw a 25% decrease in 99th-percentile latency for graph distance queries.



Comparison of 99th percentile latency for graph distance queries, control vs. setcover

For more details, check out the full [paper](http://0b4af6cdc2f0c5998459-c0245c5c937c5dedcca3f1764ecc9b2f.r43.cf2.rackcdn.com/11567-hotcloud13-wang.pdf) and [HotCloud'13 talk](https://www.usenix.org/conference/hotcloud13/using-set-cover-optimize-large-scale-low-latency-distributed-graph). The Scala implementation of this set cover algorithm is available on [github](https://github.com/linkedin-sna/norbert/blob/f9b474d12a714bbf6890ab4c2f4db8f425dc0aa7/network/src/main/scala/com/linkedin/norbert/network/partitioned/loadbalancer/DefaultPartitionedLoadBalancerFactory.scala).

# Did you mean "Galene"?

## [https://content.linkedin.com/content/dam/engineering/en-us/blog/migrated/sriramsankar_4.jpgSriram Sankar](https://engineering.linkedin.com/blog/authors/s/sriram-sankar)

Did you mean "Galene"?

*Introducing LinkedIn’s new search architecture*

**Authors:** Sriram Sankar, Asif Makhani

Search is one of the most intensely studied problems in software engineering.  It brings together information retrieval, machine learning, distributed systems, and other fundamental areas of computer science.  And search is core to LinkedIn.  Our 300M+ members use our search product to find people, jobs, companies, groups and other professional content. Our goal is to provide deeply personalized search results based on each member’s identity and relationships.

LinkedIn built our early search engines on [Lucene](http://lucene.apache.org).  As we grew, we evolved the search stack by adding layers on top of Lucene.  Our approach to scaling the system was reactive, often narrowly focused, and led to stacking new components to our architecture, each to solve a particular problem without thinking holistically about the overall system needs.  This incremental evolution eventually hit a wall requiring us to spend a lot of time keeping systems running, and performing scalability hacks to stretch the limits of the system.

Around a year ago, we decided to completely redesign our platform given our growth needs and our direction towards realizing the world’s first [*economic graph*](https://www.linkedin.com/today/post/article/20121210053039-22330283-the-future-of-linkedin-and-the-economic-graph). The result was *Galene*, our new search architecture, which has since been implemented and successfully powering multiple search products at LinkedIn. Galene has helped us improve our development culture and forced us to incorporate new development processes.  For example, the ability to build new indices every week with changes in the offline algorithms requires us to adopt a more agile testing and release process.  Galene has also helped us clearly separate infrastructure tasks from relevance tasks.  For example, relevance engineers no longer have to worry about writing multi-threaded code, perform RPCs, or worry about scaling the system.

In this post, we’ll talk about the motivations to build Galene and take a deeper look into the major design decisions of the new architecture.  If you’re a search engineer, we hope you’ll find some of our ideas useful in your own work.  Even if you don’t work on search, we hope you’ll share our appreciation of the unique problems in this space.

# The Pre-Galene Architecture

## Lucene:

Our pre-Galene search stack was centered around Lucene – an open source library that supports building a search index, searching the index for matching entities, and determining the importance of these entities through relevance scores.  Building an index is performed by calling an API that adds entities to the index.  The search index has two primary components:

·       The *inverted index* – a mapping from search terms to the list of entities that contain them; and

·       The *forward index* – a mapping from entities to metadata about them.

The lists in the inverted index are called *posting lists*.  Searching the index takes place by building a query containing search terms and then walking through the posting lists of these search terms to find entities that satisfy the query constraints.  Relevance scores are determined from details of posting list matches and from information in the forward index.

## Sensei:

Lucene indices can get quite large and often cannot be served from a single computer.  So the index is broken up into pieces (*shards*), each containing a subset of the entities.  Each shard is served from a separate computer.  With multiple shards, we require support to distribute the query to all the shards, and to manage the computers that serve the shards.

We built (and open sourced) *Sensei* to support sharding and cluster management.  There have been parallel efforts in the open source search community (eg. Solr and Elasticsearch) that build on top of Lucene to address these distributed systems challenges.

## Live Updates:

Our professional graph evolves in real time, and our search results have to remain current with these changes.  Lucene supports changes to entities by deleting the existing version of the entity, and then adding the new version.  However, when only a single inverted index term changes in an entity, we need to obtain all the other inverted index terms that map to this entity in order to create the new version of the entity.  Unfortunately, we cannot obtain this information from Lucene.  We therefore built a system called the *Search Content Store* to maintain all inverted index terms keyed by the entity.  Live updates are sent to the Search Content Store, which first updates itself, and then performs the corresponding removal and addition operations on the Lucene index.

Lucene had (until recently) another limitation with live updates – the changes to the index have to be committed before they are visible to readers of the index.  The commit process is an expensive operation and can only be performed occasionally.  To address this, we built (and open sourced) *Zoie* – which maintains an in-memory copy of the uncommitted portions of the index.  This can be used for reading until the corresponding data has been committed in the Lucene index.

## Other Components:

The pre-Galene architecture included a few additional components – *Bobo*, *Cleo*, *Krati*, and *Norbert* to name a few – to address other miscellaneous limitations of Lucene.  These components have also been open sourced.

# Hitting a Wall

As we mentioned earlier, our reactive approach to scaling the system eventually hit a wall.  We reached the limits of the pre-Galene search stack – requiring us to spend a lot of time keeping systems running, and performing scalability hacks to stretch the limits of the system.  Some of these pain points – which translated to the initial Galene requirements – are listed below:

·       **Rebuilding a complete index is extremely difficult:** Given the incremental approach to building indices, rebuilding the entire index becomes a major undertaking.  As a consequence, we minimized index improvements – thereby impacting relevance.  And in those situations where rebuilding the complete index becomes absolutely necessary – for example, when indices get corrupted – we end up with very significant investments of people’s time and effort.  We address this in Galene by moving index building to offline map-reduce jobs.

·       **Live updates are at an entity granularity:** Any updates to an entity requires inserting a new version of the entire entity and deleting the old version. This becomes an immense overhead given that entities can contain hundreds of inverted index terms.  And in addition we have to maintain a second copy of the inverted index in the Search Content Store.  In Galene, we have introduced *term partitioned segments* (described in detail later) that allow updating only the changed portions of the index, obviating the need for a Search Content Store.

·       **Scoring is inflexible:** Scoring in the pre-Galene stack is inflexible, making it very difficult to insert either hand-written or machine-learned scorers into this stack.  Scoring in Galene is performed as a separate step from retrieval and implemented as plugins.

·       **Lucene does not support all search requirements:** Lucene does not provide support for many requirements such as offline relevance, query rewriting, reranking, blending, and experimentation.  Galene uses Lucene for indexing functionality, but goes beyond to address the other search needs.

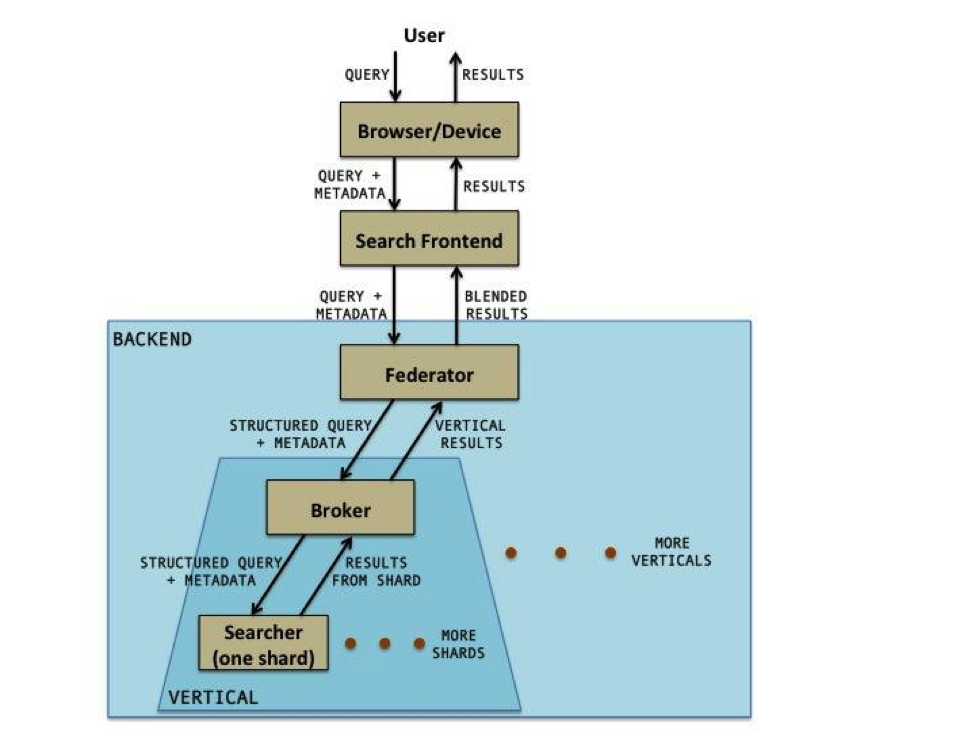
·       **Too many small open sourced components:** Given the large number of small components that were open sourced, the ownership of the overall pre-Galene system has become fragmented and spread across multiple organizations.  It has been difficult to keep these systems working together at LinkedIn.  We will not break Galene up into smaller pieces to open source separately.  Instead Galene will be a single unified framework with a single identity.

# Galene

In the new Galene architecture, we retain Lucene as the indexing layer.  We use Lucene primitives to assist in building indices, and to build queries and retrieve matching entities from the index.  Otherwise, all other functionality is outside Lucene.  Sensei, Search Content Store, Zoie, Bobo, Cleo, Krati, and Norbert have all been discarded.

## Life of a Galene Query

The following diagram shows the Galene search stack (as you can see, a typical search engine stack):



The Galene query starts at the browser/device where some processing takes place.  It then moves on to the web frontend where further processing may take place.  It then goes to the backend where the bulk of the Galene search functionality resides.  The results returning from the backend go back to the user through the frontend to the browser/device.  In this blog, we focus on the life of the query in the Galene backend – we will not discuss browser/device functionality or the frontend.

## The Federator and Broker:

The Federator and the Broker are very similar services in that they both accept a query along with metadata, and fan it out to multiple services, then wait for responses from these services, combine these responses, and return them back to the caller.

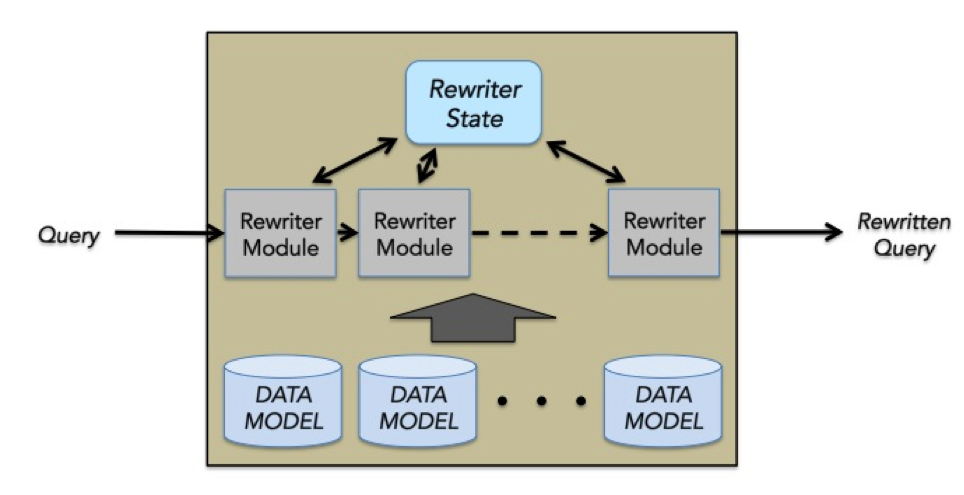
On the way down, the Federator invokes a query rewriter to rewrite the query it receives into a structured retrieval query.  The query rewriter also enhances the query with additional metadata.  The Federator then passes its output to one or more search verticals.  A search vertical serves a specific kind of entity – for example, *members*, *companies*, or *jobs*.

The receiving service in each vertical is the Broker.  The Broker may perform additional vertical specific query rewriting before passing its output to the Searchers.  The Broker waits for the Searchers to return results and then merges them together.  This merging process may be as simple as a mergesort based on a score, or could be a reranker that performs more sophisticated merging.

The merged results are sent to the Federator, which in turn combines (or blends) the results from multiple verticals.  The blending process can involve some complex relevance algorithms.  The Federator then returns the blended results to the frontend.

The Federator and the Broker are both instantiations of the same system, which offers the ability to plugin rewriters and mergers.  What makes the Federator and Broker different from each other is the actual plugins that are used.

Query rewriters are built as plugins to a rewriter API exported by the Federator and/or Broker.  The job of the query rewriter is to take the raw query and user metadata, and convert it into a structured retrieval query.  In addition, the metadata is enriched as necessary to help with the relevance measurement processes in subsequent stages.  A typical rewriter schematic is shown below:

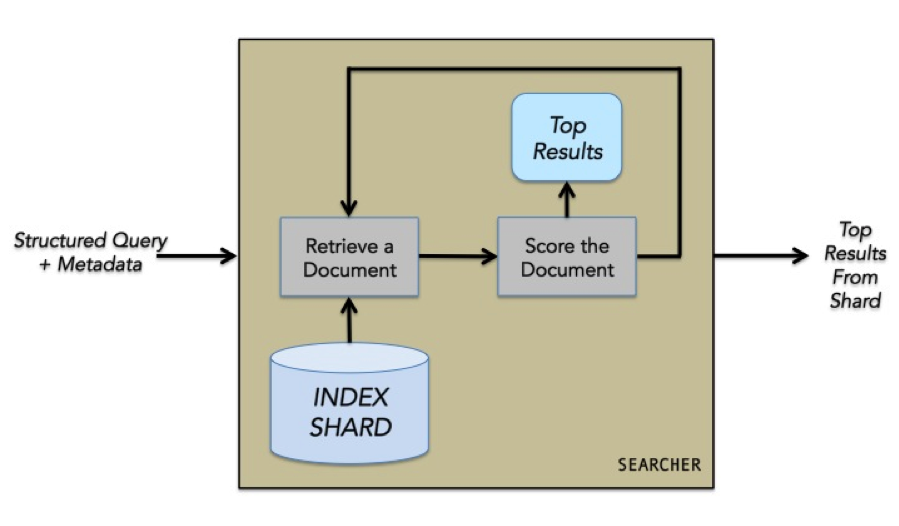


A rewriter is made up of multiple rewriter modules each of which performs a specific function. Specific functions can be synonym expansion, spelling correction, or graph proximity personalization.  Each of these rewriter modules operates in sequence and update an internal state.  After all of the rewriter modules are done, the final rewritten query is produced from this state.

The rewriter modules may need to use data models – for example, synonym maps, common ngrams, query completion data.  These data models are built offline along with the search index and copied into the Federator/Broker.

## The Searcher:

The Searcher operates on a single shard of the index.  It receives the rewritten query and metadata from the Broker, and retrieves matching entities from the index.  The entities are scored and the top scoring entities are returned to the Broker.

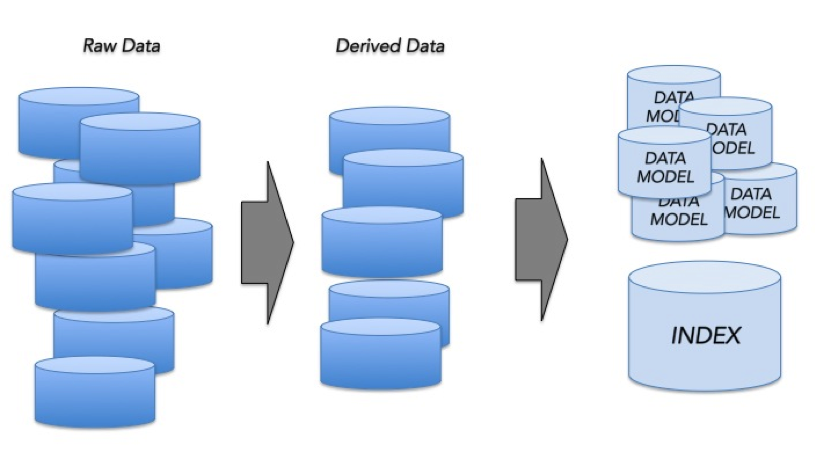


Scorers are built as plugins to a scorer API exported by the Searcher.  The scorer uses the input query, the input metadata, details on how the query matched the entity, and the forward index to determine the importance of the entity as a result for the query.  Simple scorers can be hand tuned, but more sophisticated scorers are built semi-automatically using a machine learning pipeline.

## Indexing on Hadoop:

Indexing on Hadoop takes the form of multiple map-reduce operations that progressively refine the data into the data models and search index that ultimately serve live queries.  HDFS contains raw data containing all the information we need to build the index.  We first run map reduce jobs with relevance algorithms embedded that enrich the raw data – resulting in the derived data.  Some examples of relevance algorithms that may be applied here are spell correction, standardization of concepts (for example, unifying “software engineer” and “computer programmer”), and graph analysis.

Galene provides custom map-reduce templates that perform the final step of building the actual index and data models.  These templates are instantiated for specific jobs through schema definitions.



## Early Termination:

Galene provides the ability to assign a *static rank* to each entity.  This is a measure of the importance of that entity independent of any search query, and is determined offline during the index building process.  Using the static rank, we order the entities in the index by importance, placing the most important entities for a term first.  The retrieval process can then be terminated as soon as we obtain an adequate number of entities that match the query, and not have to consider every entity that matches the query.  This strategy is called *early termination*.

Early termination works if the static rank of an entity is somewhat correlated to its final score for any query.  The main benefit of early termination is performance – scoring is usually an expensive operation and the fewer the entities scored the better.  Looked at in another way, early termination allows us to use more sophisticated scorers.

To maximize the benefit of early termination, the query rewriting process should bias the query towards retrieving the most relevant entities.

## Live Updates:

Live updates in the pre-Galene architecture had a few significant problems:

·       We have to make updates at the granularity of an entity, which impacts performance

·       We have to maintain a second copy of the index in the Search Content Store

·       Adding and removing entities from the index upsets the static rank order and the ability to perform early termination

·       The index is always being modified resulting in a brittle system making it difficult to easily recover from index corruptions, etc.

In Galene, live updates are performed at the granularity of single fields.  We have built a new kind of index segment – the *term partitioned segment*.  The inverted index and forward index of each entity may be split up across these segments.  The same posting list can be present in multiple segments and a traversal of a single posting list becomes the traversal of a disjunction of the posting lists in each of the segments.  For this to work properly, the entities in each segment have to be ordered in the same manner - given that we order entities by static rank in all segments, we satisfy the ordering constraint.  The forward index becomes the union of the forward indices in each of the segments.

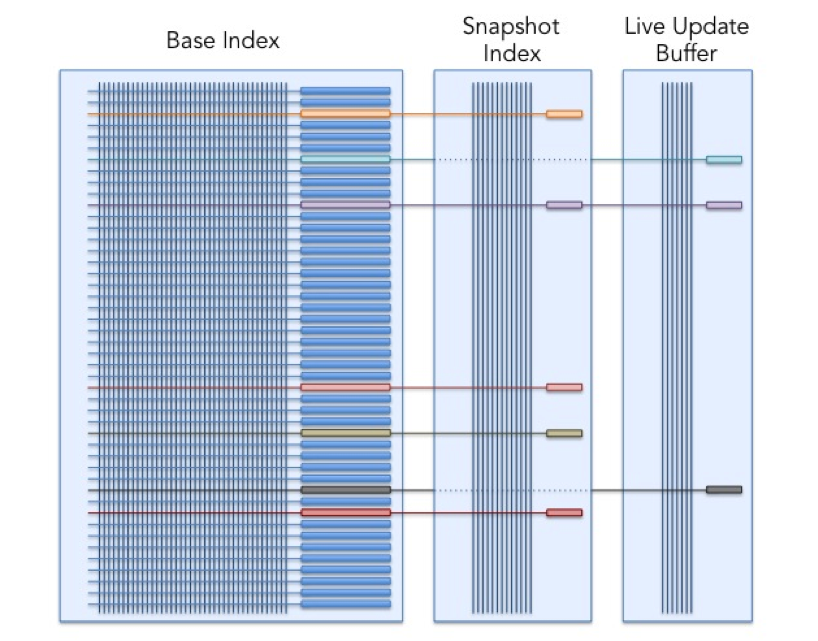
In Galene, we maintain three such segments:

·       The *base index* – this is the one built offline on Hadoop.  This is rebuilt periodically (say every week).  Once built, it is never modified, only discarded after the next base index is built.

·       The *live update buffer* – which is maintained in memory.  All live updates are applied to this segment.  This segment is designed to accept incremental updates and augment itself to retain the entities in the correct static rank order.

·       The *snapshot index* – given that the live update buffer is only in memory, we periodically (every few hours) flush it to the snapshot index on disk to make it persistent.  If the snapshot index already exists, a new one is built that combines the contents of the previous snapshot index and the live update buffer.  After each flush, the live update buffer is reset.

This multi-segment strategy is illustrated below.  The horizontal lines are entities and the vertical lines are posting lists.  The box at the right extreme of each entity represents its forward index.



## Index Lifecycle Management:

Our indexing strategy is complex – a base index is built every week, snapshots are generated every few hours, and these indices have to be present on all replicas of the searchers.

Generating snapshots is a costly operation.  We cannot afford to do this on the searcher machines.  Instead we have another set of machines called *indexers*.  Indexers are identical to searchers in capability – except that they do not serve search traffic (they only accept live updates).  Every few hours, indexers merge their live update buffer with their snapshot index.  The snapshot indices then get shipped to all the corresponding searchers.

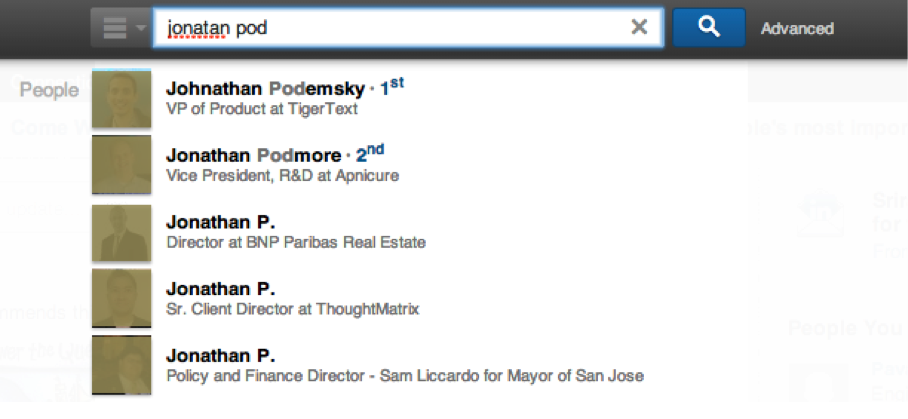
To support this we need an efficient and convenient mechanism to move the indices from where they are generated to where they are used – i.e., the base index needs to be moved from HDFS to the searchers and indexers, and the snapshot index from the indexers to the searchers.

We have built a bit torrent based framework to address this.  This framework defines a concept called the *replica group*.  Machines can join replica groups and automatically get all data associated with that replica group.  These replica group members can also add data to their replica groups, which then get replicated to all the other members.

This framework is also used to move data models from HDFS to the Federators and Brokers.  Additional lifecycle management, such as versioning of indices and rolling back capabilities, are also built into this system.

# Galene in Action:  Instant Member Search

One of the applications currently powered by Galene in production is Instant. Instant is the typeahead search experience that enables you to navigate and explore LinkedIn instantly. The most common use-case is Instant Member Search which provides the ability to search all of our 300M+ members by name.



We were able to provide significant improvements to the pre-Galene implementation that we replaced:

·       **Searching all of LinkedIn:**  Previously, Instant Member Search was limited to first and second-degree connections due to various constraints in the older architecture.  With Galene, members can perform typeahead searches that include all of our 300M+ members.

·       **Better relevance:**  Galene’s Instant Member Search has a more sophisticated relevance algorithm that includes offline static rank computation, personalization through factors such as connection degree, and approximate name matching.  Previously, it was not possible to incorporate such relevance functionality.

·       **Faster and more efficient:**  The new Instant Search is more than twice as fast as the previous implementation, utilizing about a third of the hardware – while still providing the gains described above.

# Going Forward

The bulk of the work towards building Galene is complete, and we are realizing benefits today through powering multiple verticals in production.  Our focus going forward is to continue migrating search verticals into Galene, while at the same time improving the quality of each individual search experience.

Galene has helped us improve our development culture and forced us to incorporate new development processes.  For example, the ability to build new indices every week with changes in the offline algorithms requires us to adopt a more agile testing and release process.  Galene has also helped us clearly separate infrastructure tasks from relevance tasks.  For example, relevance engineers no longer have to worry about writing multi-threaded code, perform RPCs, or worry about scaling the system.

As we migrate more complex search products, we continue to evolve the Galene infrastructure to make it even more versatile.  Some of the areas we are currently working on are highlighted below.

## Improved Relevance Support:

Work is underway to enhance the scorer plugins to support machine-learned models (gathering training data, training, and deployment of models) in a systematic manner.  The rewriter plugin framework is also being improved to become a more complete query planning framework.  This framework will allow some iterative processing (for example, the ability to reissue a misspelt query after a second round of more expensive spell corrections when the first round did not return adequate results), and some composition capabilities (for example, the ability to build queries that depend on the results of other queries).

## Search as a Service:

The search platform is being used to power multiple applications and products at LinkedIn – ranging from member-facing products such as People Search, Job Search and LinkedIn Recruiter to internal applications such as customer-service and advertising support.   There is a growing list of applications at LinkedIn that require search functionality.

In order to scale across these search use cases, we are enhancing our cluster management system (inherited from Sensei) to automatically scale with data and traffic, provide the necessary system monitoring, provision hardware automatically, as well as provide seamless software upgrades across all applications.

Once this capability is ready, it should be possible for a simple search based application to be up and running in hours primarily through the specification of configuration schemas.  All complexities associated with scheduling index builds, hardware allocation and scaling for capacity will be abstracted away.

## Exploring the Economic Graph:

Sophisticated search functionality needs sophisticated interconnected data – provided by LinkedIn’s professional graph. Going forward, we have ambitious plans to further enrich this data by incorporating all the economic data there is in the world – to obtain the world’s first *economic graph*.

The Galene search architecture has been designed with this long-term vision in mind.  Index terms do not have to be terms actually present in the entities.  They can be attributes – such as graph edges.  In fact, our Instant Member Search vertical already indexes a few different kinds of edges to help in providing social signals for relevance.

The query “**group:82282 AND company:1337 AND term:information AND term:retrieval**” exemplifies the kinds of capabilities that we are working on to support future search products.  82282 is the id for the *Lucene Users* group and 1337 is the id of *LinkedIn* (the company).  The query retrieves members of the Lucene Users group that work at LinkedIn and have mentioned “information retrieval” in their profile.  Other economic graph queries could be:

·       Engineers with Hadoop experience in Brazil

·       Data science jobs in New York in companies where my connections have worked at

·       Connections of Asif or Sriram who work at Google

# In Conclusion . . .

The past year has been immensely fruitful as we are moving to a more agile development process with the potential of frequent relevance improvements.  There are other systems within LinkedIn that are based on the Lucene/Sensei stack and facing similar scaling issues – including the recommendations engine, newsfeed, and ads ranking.  As we migrate search verticals to Galene, we will also work on migrating these other systems to Galene.

We would like to acknowledge the immense contributions of the search team.  While we served as the architect (Sriram) and manager (Asif) of the Galene migration effort, we could not have done it without a stellar team that worked extremely well together and came up with many innovations.  The team and the product efforts have complemented each other – resulting in not only a great product, but also a great cohesive team that we are proud to be a part of.

**Thank you team, for this wonderful journey!**



**Memcached** (pronunciation: mem-cash-dee) is a general-purpose distributed [memory caching](https://en.wikipedia.org/wiki/Memory_caching) system. It is often used to speed up dynamic [database](https://en.wikipedia.org/wiki/Database)-driven websites by caching data and [objects](https://en.wikipedia.org/wiki/Object_%28computer_science%29) in [RAM](https://en.wikipedia.org/wiki/Random-access_memory) to reduce the number of times an external data source (such as a database or API) must be read. Memcached is [free and open-source software](https://en.wikipedia.org/wiki/Free_and_open-source_software), licensed under the [Revised BSD license](https://en.wikipedia.org/wiki/Revised_BSD_license).[[2]](https://en.wikipedia.org/wiki/Memcached#cite_note-Memcached_license-2) Memcached runs on [Unix-like](https://en.wikipedia.org/wiki/Unix-like) operating systems (at least [Linux](https://en.wikipedia.org/wiki/Linux) and [OS X](https://en.wikipedia.org/wiki/OS_X)) and on [Microsoft Windows](https://en.wikipedia.org/wiki/Microsoft_Windows). It depends on the [libevent](https://en.wikipedia.org/wiki/Libevent) library.

Memcached's [APIs](https://en.wikipedia.org/wiki/Application_Programming_Interface) provide a very large [hash table](https://en.wikipedia.org/wiki/Hash_table) distributed across multiple machines. When the table is full, subsequent inserts cause older data to be purged in [least recently used](https://en.wikipedia.org/wiki/Least_recently_used) (LRU) order.[[3]](https://en.wikipedia.org/wiki/Memcached#cite_note-3)[[4]](https://en.wikipedia.org/wiki/Memcached#cite_note-4) Applications using Memcached typically layer requests and additions into RAM before falling back on a slower backing store, such as a database.

**Couchbase Server**, originally known as **Membase**, is an [open-source](https://en.wikipedia.org/wiki/Open-source), distributed ([shared-nothing architecture](https://en.wikipedia.org/wiki/Shared-nothing_architecture)) [multi-model](https://en.wikipedia.org/wiki/Multi-model_database) [NoSQL](https://en.wikipedia.org/wiki/NoSQL) [document-oriented database](https://en.wikipedia.org/wiki/Document-oriented_database) software package that is optimized for interactive applications. These applications may serve many [concurrent users](https://en.wikipedia.org/wiki/Concurrent_user) by creating, storing, retrieving, aggregating, manipulating and presenting data. In support of these kinds of application needs, Couchbase Server is designed to provide easy-to-scale key-value or JSON document access with low latency and high sustained throughput. It is designed to be [clustered](https://en.wikipedia.org/wiki/Cluster_%28computing%29) from a single machine to very large-scale deployments spanning many machines. A version originally called **Couchbase Lite** was later marketed as Couchbase Mobile combined with other software.

Couchbase Server provided client protocol compatibility with [memcached](https://en.wikipedia.org/wiki/Memcached),[[2]](https://en.wikipedia.org/wiki/Couchbase_Server#cite_note-2) but added disk [persistence](https://en.wikipedia.org/wiki/Persistence_%28computer_science%29), [data replication](https://en.wikipedia.org/wiki/Data_replication), live cluster reconfiguration, rebalancing and [multitenancy](https://en.wikipedia.org/wiki/Multitenancy) with [data partitioning](https://en.wikipedia.org/wiki/Partition_%28database%29).

# Kafka at LinkedIn: Current and Future

## [https://content.linkedin.com/content/dam/engineering/en-us/blog/migrated/Mammad%20Zadeh.jpgMammad Zadeh](https://engineering.linkedin.com/blog/authors/m/mammad-zadeh)



The LinkedIn engineering team has developed and built [Apache Kafka](http://kafka.apache.org/) into a powerful open source solution for managing streams of information. We use Kafka as the messaging backbone that helps the company’s applications work together in a loosely coupled manner. LinkedIn relies heavily on the scalability and reliability of Kafka and a surrounding ecosystem of both open source and internal components. We are continuing to invest in Kafka to ensure that our messaging backbone stays healthy as we ask more and more from it.

## Use Cases at LinkedIn

Today, some of the common scenarios at LinkedIn that leverage Kafka include:

1. **Monitoring :** All hosts at LinkedIn emit metrics pertaining to their system and application health through Kafka. These are then collected and processed to create monitoring dashboards and alerts. A deeper read on this can be found [here](https://engineering.linkedin.com/52/autometrics-self-service-metrics-collection). In addition to standard metrics, a richer [call graphs analysis](http://engineering.linkedin.com/samza/real-time-insights-linkedins-performance-using-apache-samza) is also done by leveraging [Apache Samza](http://samza.incubator.apache.org/) for processing the events in real time.

2. **Traditional messaging:** Various applications at LinkedIn leverage Kafka as a traditional messaging system for standard queuing and pub-sub messaging. These applications range from Search, Content Feed and Relevance and they publish processed data into online data serving stores like Voldemort, etc.

3. **Analytics:** LinkedIn tracks data to better understand how our members use our products. Information such as which page got viewed and which content got clicked on are sent into a Kafka cluster in each data center. These events are all centrally collected and pushed onto our Hadoop grid for analysis and daily report generation.

4. **As a building block (log) in various distributed applications/platforms:** Kafka is also leveraged as a core building block (distributed log) by other products like our big data warehousing solution [Pinot](https://engineering.linkedin.com/analytics/real-time-analytics-massive-scale-pinot). We are also working on using Kafka as an internal replication and change propagation layer for our distributed database Espresso.

## ****Kafka Ecosystem at LinkedIn****

Apache Kafka has to be augmented with a set of components to enable several usage scenarios. At LinkedIn, the Kafka ecosystem comprises of the following set of components in addition to Apache Kafka.

**1.** [**MirrorMaker**](https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=27846330) : This is an open source project that is used to move data between Kafka Clusters. In many situations we need the business logic to operate on events that are being generated in multiple data centers. MirrorMaker is used to aggregate these events across data centers.

**2. A REST interface:** This interface enables non-java applications to easily publish and consume messages from Kafka using a thin client model.

**3. A schema registry:** At LinkedIn we have, for the most part, standardized on [AVRO](http://avro.apache.org/) for the event schemas. We have a layered API to send and receive AVRO events on top of the core KAFKA APIs. This API implicitly uses a Schema Registry service to serialize and deserialize the events as they are sent and received from Kafka.

**4. Auditing service** : Events get generated in one LinkedIn data center. However, they typically get moved to a different data center for a lot of the offline processing. During this move, it is important for the consuming application (e.g. map reduce jobs) to understand when it has received all events that were generated in a particular time window, so that it can begin the offline processing. The audit service, which is also built on top of Kafka, is used to solve this problem.

**5. A bridge to push data from Hadoop into Kafka** : Most of the data derived in our map-reduce clusters (Hadoop) is pushed back to our serving databases (like Voldemort) using this bridge that pushes data from Hadoop into Kafka.

## ****The Future of Kafka at LinkedIn****

As LinkedIn scales, we know we need to stay ahead of the growth curve and ensure that our messaging backbone stays healthy. As the company grows, we expect the number of teams and applications that use Kafka and the diversity of their needs to increase. Also, as the number of users (developers and site reliability engineers) increases we expect to see a bigger delta in how well the users of Kafka understand Kafka. As a result we need to make sure that simple errors can be avoided.

We are planning to focus on the following key areas of development in 2015:

**1. Security** : We need to add basic authentication and authorization capabilities to the Kafka broker. This has to cover both management and runtime operations. This is necessary for certain types of data even within a secure network to avoid human errors. A lot of this work has already started in the open source community.

**2. Quotas** : Given that we have a wide variety of applications leveraging the same Kafka cluster it is important to make sure that one application doesn’t accidentally end up using all of the system resources and negatively impact all of the applications on the Kafka cluster. In Kafka we have to worry about saturating the network card on each broker host and also top of rack switches. When applications start reading from the beginning of the Kafka log or just catching up after a prolonged pause, they can saturate the network. This type of situation can cause unintended consequences for a lot of other applications sharing the network. This will be an important area of focus for us.

**3. Reliability and Availability** **:** We pick up Kafka from the open source trunk. In the open source, there are a good number of contributors. As in any project with a lot of engineers a certain degree of rigor is required in terms of upgrade, failover and stress testing before we roll the bits out to production. We will continue to invest in this area. In addition, a lot of time is spent in finding and fixing issues that are found in our day to day usage of Kafka.

**4. Core functionality** **:** Over the past year we created a [new set of APIs](https://cwiki.apache.org/confluence/display/KAFKA/Client+Rewrite) in Kafka to allow pipelining publish operations and hence better performance for publishing events into Kafka. Over the course of the next year we hope to also work towards a new consumer API. The new consumer API will essentially remove the dependency on Zookeeper for the Kafka client. This also enables us to have a complete security model.

In addition to these improvements to our thick client, we are investing heavily on our thin client for Kafka. Basically we want to make the REST interface to Kafka be first class with support for core set of SLAs and monitoring.

**5. Being Cost Efficient:** As the company scales it is obviously important to make sure that the Kafka broker clusters can scale. We currently reach up to 550 Billion events a day while reaching peaks of 8 million incoming messages a second and 32 million outgoing messages a second. Kafka is designed for this. There is, however, work that needs to be done to ensure that we don’t keep solving the scale problem by throwing more and more hardware at the problem.

**6. New Initiatives:** We have some new initiatives in our data infrastructure group, which would leverage Kafka in new scenarios. Specifically, our distributed database [Espresso](http://data.linkedin.com/projects/espresso) will start leveraging Kafka for replication between the primary Espresso storage nodes and their secondaries. Currently, LinkedIn mostly uses Kafka for throughput. These additional use cases are going to also require lower latencies. We will also have to measure the failover time for Kafka broker primaries a bit more closely and have to potentially tune it.

**7. Improving Operability:** We currently have many manual processes with Kafka. Our site reliability engineers (SREs) have created a lot of cool tools to help with this. But we need to get some of these things solved in Kafka itself. For example, when we move partitions we have to be careful not to saturate the network. Today this is done carefully by SREs. We need the software to take care of this to reduce human error. When a node is added it would be great if Kafka automatically moves the appropriate set of partitions while keeping the cluster balanced.

## Collaboration with Open Source Community

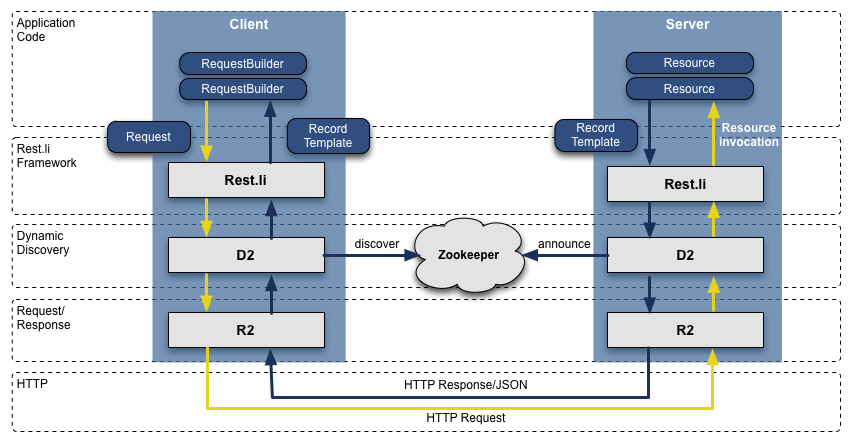
A few months ago some of the original engineers on Apache Kafka departed LinkedIn to start [Confluent](http://confluent.io/), a startup focused on Kafka. We wish them well and are excited about the broadening of the Kafka community. We will continue to collaborate and work closely with all contributors to Apache Kafka to take it to the next level.

# Rest.li: RESTful Service Architecture at Scale

## [https://content.linkedin.com/content/dam/engineering/en-us/blog/migrated/joeBetz..jpgJoe Betz](https://engineering.linkedin.com/blog/authors/j/joe-betz)

Today we are announcing the open-sourcing of Rest.li, a piece of infrastructure developed and used here at LinkedIn. Rest.li is a REST+JSON framework for building robust, scalable service architectures using dynamic discovery and simple asynchronous APIs. We feel that Rest.li fills a niche for building RESTful service architectures at scale, offering a developer workflow for defining data and REST APIs that promotes uniform interfaces, consistent data modeling, type-safety, and compatibility-checked API evolution.

Rest.li was created to deal with a number of problems at LinkedIn. First, we needed a standard way to describe the resources available in our service-oriented architecture and enable access by diverse clients written in any language. We wanted to focus on standardizing common API operations, while still allowing developers to create non-standard operations. Secondly, we needed our solution to be scalable and asynchronous. Many of our services receive several thousand queries per second per instance, and we needed our solution to work under that sort of load. We wanted to ensure that writing APIs would be fast and easy for our developers, even if they are not well-versed with REST. We also needed our system to support long-term evolution and growth of our interfaces.

  
**Figure 1**. Rest.li Client Server Flow

Rest.li is a Java framework for building REST-style services using well known concepts such as an entity-oriented design, standard HTTP-style verbs to operate on those entities, and a flexible serialization scheme for entities. We've added standard batch operations to support efficient access to large data sets. It also includes Java client libraries and supports type-safe access to resources and entities on both the client and server side. Rest.li also specifies an IDL for describing resources.

R2 is a REST transport layer abstraction in Java. It exposes a low-level REST-style interface inspired by HTTP and a high-performance asynchronous API. We use Netty to build R2's HTTP implementation. R2 has also allowed us to experiment with mapping REST operations to non-HTTP transports.

D2 is our dynamic discovery and client-side load balancing layer. We use [ZooKeeper](http://zookeeper.apache.org/) as a registry for information about available services and the hosts that provide them. Clients automatically get the latest information from ZooKeeper and apply load-balancing algorithms on the client side to distribute load evenly across servers and reduce load to overloaded servers.

We've been using Rest.li for well over a year at LinkedIn. All our new services are built using it and we've converted many of our pre-existing services over. We think it's so important to have a uniform set interfaces that represent our data that we're aggressively migrating all, yes all, our services to Rest.li. And we're already well into this transition with many core services that power our site using Rest.li, including people following, our recommendation engine systems and the network update stream on the homepage.

We hope every Java developer working with RESTful APIs will give Rest.li a try. While a big benefit of Rest.li is that it can scale to the needs of world class data centers, it's designed to be lightweight, approachable and to foster REST best practices. This makes it a great way to get a RESTful API up and running for projects both small and large, even if you're not a REST expert.