Designing UberLite: a Ride Aggregator Service*

Guna Prasaad
University of Washington
guna@cs.washington.edu

Johan Per Vikstrom
University of Washington
pjvik@cs.washington.edu

1 INTRODUCTION

Up until two decades ago, the most important "technical" decision made by executives in a newly formed business was *which enterprise-grade database software to purchase for their IT needs?* While they still make that decision, gone are the days when a single omnipotent database was sufficient to get a business up and running. The internet has revolutionized the way businesses are built and operated. Most internet-based applications today are designed for deployment at scale, which is often beyond the bounds of a single complicated piece of database software. Hence, companies today compose a plethora of services that together support the various features of an user-facing application.

Micro-services is an architectural style that composes a large system using a collection of services, where each (1) has a well-defined specification that is easy to build, test, deploy and maintain independently; (2) can communicate with other micro-services using a pre-defined API; and (3) owned by a small team of engineers who are responsible for its overall health and performance. Needless to say that each of these micro-services are - more often than not - distributed systems themselves, potentially replicated across global locations.

In this report, we aim to analyze how a complex application can be decomposed into micro-services, pick the right set of tools (open-source or otherwise) to build them and analyze the various trade-offs involved at every stage. In order to keep our exploration bounded, we will focus on one application in the form of a case study. In this report, we seek to design an Uber-like ride-aggregator service, called UberLite. We wish to answer the following questions:

- What features does UberLite support?
- What are the high-level business objectives of UberLite, and how do these impact the architecture of the system?
- How can we decompose UberLite into simpler micro-services?
- How do the goals of UberLite translate into that of each micro-service, in terms of performance, consistency and availability guarantees?
- What does the workload for each micro-service look like? How should the system handle temporary skews in the workload?
- What is an *ideal* specification for the micro-services? What is practical and why?
- How can we build these micro-services? What are some open-source tools that can be used to build them? How can they be compared? Why choose one over another?
- What are the limitations with the presented design?

The remainder of the report is structured as follows: Sec. 2 provides a brief background on ride-aggregator services and presents the key business challenges in UberLite; Sec. 3 describes the anatomy of a trip within UberLite and how a trip interacts with the services within UberLite; Sec. 4 describes the core UberLite services - from booking to the actual ride and payment; In Sec. 5, we talk about the background services that feed into and power the core services; Sec. 6 details our recommendations for implementing the core and background services using open-source tools; In the last section, Sec. 7, we attempt to summarize the lessons learned over the course of working on the project.

Disclaimer: This report does not intend to propose a new or novel, original micro-service architecture for UberLite, but rather to study how one goes about the design process. We thank Uber for sharing technical discussions and design decisions on their engineering blog (https://eng.uber.com/), which we have used extensively to inspire the discussion in this report.

2 BACKGROUND

Ride aggregator services like UberLite are the quintessential two-sided marketplace [3], that connects riders to drivers.

The main advantage of such a service is its *efficiency*. The traditional cab system operates through forced scarcity — there can only ever be a certain number of cabs on the road in any given city. As a result, fares are high. Cabs are not around when you need them, like late at night or in the pouring rain. Old-school taxis find fares either by driving around and picking them off the street (street hail), or having a dispatcher (with phoned-in customer requests) instructing them on where to go. Both these methods have significant limitations [3]:

- *Poor supply allocation for drivers.* Traditional cabs only have a passenger in the car 30–50% of the time. To find a fare, they're dependent on taxi stands, centralized dispatchers, or being hailed from the street — they have no other way to know where to go next.

*This report was prepared as part of a graduate distributed systems course (CSE 552) project at the University of Washington in Fall 2019.

- *Low supply liquidity for riders.* Traditional cabs congregate in urban cores and high-transit areas, leaving outer boroughs, suburbs, and "less profitable" areas under served. Riders in these areas are often unable to get a cab at all, or face long waits.

With UberLite-like services, on the other hand, users request rides directly through the app. The nearest driver is dispatched to their location, and they can be hailed again immediately after drop-off.

The core value of the UberLite marketplace is reliability. The fundamental business problem is balancing supply and demand, both temporally and spatially. Uber uses *dynamic pricing* to adjust supply of drivers and demand for rides. When UberLite increases the price for rides in a region, it

- Incentivizes drivers to operate when/where there is higher surge price due to higher returns (more drivers).
- Dis-incentivizes riders who do not want to pay a premium. They either look for alternate means of transportation or wait until surge price is turned off (less riders).

So the reliability metric of UberLite as a transportation service is how well we can fulfill the demand for rides. Which means UberLite must be able to distribute supply and demand spatio-temporally. This becomes problematic because rides can be long, we don't know where riders will appear before they do and big events/airports will cause big spikes in demand. So UberLite must use the realtime data about rides and historical data to dynamically adjust the price to increase supply and fulfill as many rides as possible.

UberLite matches riders to drivers in the neighborhood. For each such potential (rider, driver) pairs it computes several features: distance to pickup, distance of the ride, the time to pickup and the time to drop-off. While this could be fairly straight-forward, just take a distance estimation and multiply by the speed, the actual time-estimate also depends on realtime monitors such as the traffic in the route. Using this information UberLite arrives at a price estimate that takes time, distance and the surge price multiplier into account.

$$p = ((c_d * d + c_t * t) * s + c_m) * \eta * (1 + T)$$

where c_d is the cost per unit distance, d is the distance, c_t is the cost per unit time, t is the estimated time, s is surge price multiplier for the pickup location, c_m are miscellaneous cost such as toll-fees, η is a discount rate based on different promotions and such and T is the service tax in the region.

Once all these parameters have been computed for every (driver, rider) pair in the batch we pick a *good matching*, where a good matching is defined based on business goals, and we do not delve into that here.

3 ANATOMY OF A TRIP IN UBERLITE

In this section, we provide a brief overview of the anatomy of an UberLite trip. A trip in UberLite is modeled as an asynchronous state-machine (refer Fig. 1). The state is realized by a trigger-based framework allowing the services to subscribe to changes on the state. The services can then update the ride-state without having to know about what should happen in the next state.

The state machine for a trip works as follows (refer Fig. 1).

- A trip state begins when the rider requests a ride.
- Request is posted to the core services (Sec. 4).
- Core marketplace services estimates the time, cost for the ride and a potential driver for the ride and writes the assignment back onto the store.
- The rider is presented with the cost, which is either accepted or rejected.
- Once rider accepts, the ride is proposed to the driver. The driver can either accept or decline the ride.
- If the driver declines the ride or doesn't respond in a timely manner (i.e. times out) and there have been fewer than k attempts at driver assignment, UberLite retries by posting the request back on the Marketplace.
- If the driver declines the ride and there have been k attempts, we declare unsuccessful and notify the rider.
- If the driver accepts the ride, then the trip state machine moves to driver en-route to pickup.
- Once driver picks up the rider, the driver notifies the system - updating the state to rider picked up.
- When the ride is complete and rider dropped off, the driver notifies the system again.
- Once payment is fulfilled the state machine ends.

4 CORE SERVICES

We now discuss the core services within UberLite and how they come together to power ride-matching in realtime.

4.1 Functional Specification

We briefly describe the functional specifications of the core services below, and Fig. 4 depicts the data flow among them.

Map Indexing Service. Geo-locations within UberLite are specified using a geo-spatial index called H3 [28] that was developed and open-sourced by Uber. H3 is a hexagonal hierarchical spatial index. It partitions the globe into a hierarchy of hexagons as shown in Fig. 2. It assigns a unique hierarchical index to each cell.

The H3 index of a resolution r cell begins with the appropriate resolution 0 base cell number. This is followed by a sequence of r digits $0-6$, where each i^{th} digit d_i specifies one of the 7 cells centered on the cell indicated by the coarser resolution digits d1 through d_{i-1} as shown in Fig. 3. A local hexagon coordinate system is assigned to each of the resolution 0 base cells and is used to orient all hierarchical indexing child cells of that base cell. The assignment of digits $0-6$ at

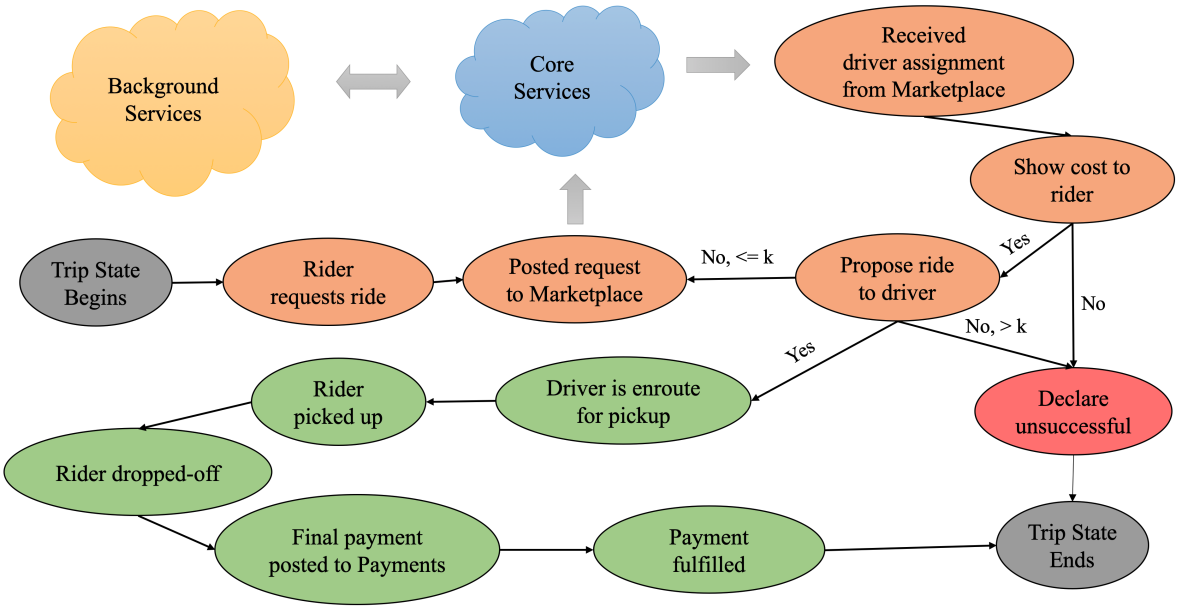
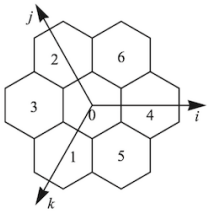


Fig. 1. State machine for a trip maintained in the Trip Service

Fig. 2. H3 index divides a geographical region into a hierarchical hexagonal grid.



octal	0	1	2	3	4	5	6
ijk+	(0, 0, 0)	(0, 0, 1)	(0, 1, 0)	(0, 1, 1)	(1, 0, 0)	(1, 0, 1)	(1, 1, 0)
binary	000	001	010	011	100	101	110

Fig. 3. H3 CPI Scheme [28]

each resolution uses a *Central Place Indexing* arrangement. This indexing scheme is key to how UberLite organizes and operates on map data.

Driver Discovery Service (DRS). The driver discovery service maintains the current location of all *active* drivers in the marketplace in realtime. The primary role of DRS is to identify potential drivers for a ride, given the location of a rider. Finding drivers in the "neighborhood" can be decomposed into a set of range queries on the H3 index. Location of active drivers are constantly monitored and updated in realtime - the driver client sends the current GPS location periodically to the service.

Route Service (RS). Route service is responsible for finding potential routes from point A to point B. More specifically, given the current location of the driver, pick up and destination, RS figures out k potential routes for the driver, both for the pickup and the actual ride. We find more than one route since the shortest distance route may not be the shortest one in terms of time.

Time Estimation Service (TES). TES has two specific functions: (1) Monitor traffic on the roads within the area of service and maintain a heat map; (2) Use the heat map to estimate time to perform a ride given the route.

The heat map specifies a hotness parameter α, ($\alpha \geq 1$) that is a time multiplier used to estimate time taken to cover a given stretch of the road with traffic. This heat map can either be maintained in-house or could be bought directly from other map service providers such as Google.

TES also estimates the time taken to get from point A to point B via a given route. TES uses the traffic information to estimated time to pickup and drop-off. It is very important to make an accurate time estimation the cost estimate for the ride depends on estimated time as mentioned in Sec. 3.

Surge Pricing Service (SPS). As mentioned earlier, dynamic pricing is key to balancing supply and demand spatio-temporally in UberLite. SPS is the micro-service endpoint to obtain surge pricing multiplier for a given hexagon index. Surge pricing multipliers are updated in realtime based on supply-demand in every neighborhood. This endpoint simply queries the pre-computed values.

Discounts and Promotions Service (DPS). UberLite provides a variety of discounts and promotions for riders as well as drivers to encourage driving or taking rides. This information is provided by the discounts and promotions service. These values are pre-computed on a daily basis.

Tax & Tolls Service (TTS). Maintains the service tax details for each city, state, etc. that is queried for cost estimations. Also maintains tolls or other miscellaneous fee information that is specific to a route. Specifically given a route, it returns the total accumulated cost of such miscellaneous fee.

Price Estimation Service (PES). The price estimation service computes estimated cost for a given route and rider, driver pair using the formula specified in Sec. **??**. This service communicates with all other services to obtain relevant information to compute the estimated cost.

Matching Service (MS). The matching service takes in a set of (rider, driver, route) triplets and computes the best driver route matching for the set of riders. It takes a feature vector for each (rider, driver, route) triplet that includes all relevant information about the match.

4.2 Analysis

We now analyze (Table 1) the access patterns (update/read traffic), consistency and computational requirements of these core services based on the following parameters.

- *Stateful vs. stateless.* Is the service stateful or stateless?
- *Update traffic.* How frequently is the state associated with the service updated? Does that update scale with number of riders/drivers?
- *Read/Query traffic.* How frequently is the service queried (both for stateful and stateless services)? Does it scale with the number of riders/drivers?
- *Consistency.* Can we operate with stale data? Do we need higher levels of consistency?
- *Compute size.* Is the query computationally expensive?
- *State size.* How large is the state? Does it scale with the number of riders/drivers?

5 BACKGROUND SERVICES

We now describe essential background service that power UberLite core services with required data for seamless functioning. We summarize the data and compute requirements for background services in Table 2.

Forecasting. Since dynamic pricing plays a central role in sustainable functioning of UberLite, being able to predict supply and demand is mission-critical. Note that this forecasting must be performed for each neighborhood at each unit of time. This distribution over time determines the surge price multiplier that balances supply-vs-demand.

Forecasting dependents on several factors. Firstly, there is a steady demand for rides on weekdays from people who use UberLite for everyday commute. While this is reasonably easier to forecast, demand can also be seasonal. For instance, there is a higher demand for rides on a New Year or Independence Day to and from public places such as parks. Even more specific events can skew demand towards a particular time and space - for example, the end of a Seahawks game at the UW Stadium. More so, even realtime events such as a change in weather can affect the forecast - demand for rides are known to be higher when it is raining since public transport is less preferable.
Essentially, the forecasting service must learn from:

- Historic supply vs. demand data
- Event aggregators
- Weather reports
- Realtime changes in supply vs. demand

This service requires both offline training and online predictions involving machine learning and data analysis.

Pricing. Given a forecast for driver supply and rider demand, the Pricing Service forecasts the surge pricing to be applied for each neighborhood. Since the forecast can change in realtime, the pricing scheme also has to operate in realtime. The pricing service is also used to send periodic (say every 4 hours) alerts or notifications to drivers specifying that surge pricing is on at a particular time and location. These notifications help UberLite to prepare in advance for an increase in demand. Designing appropriate algorithms

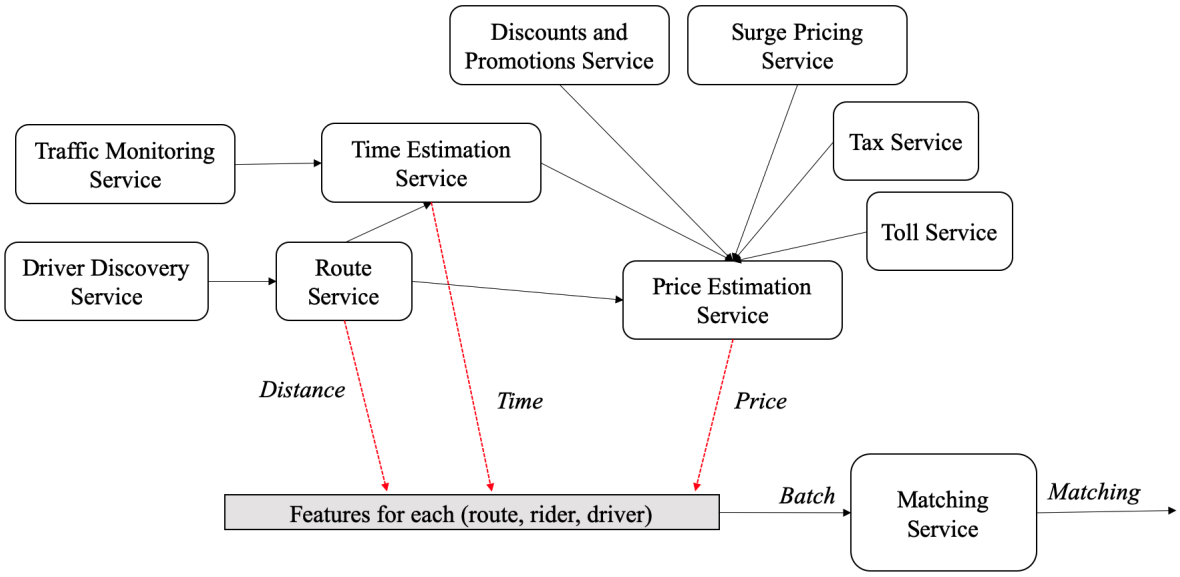


Fig. 4. Putting all Core Marketplace Services together

Service Name	State				Queries	Compute
	Stateful	State size	Updates	Staleness		
Driver Discovery	Yes	$O(D)$	Realtime	Severe	$O(R)$	Low
Traffic Monitoring	Yes	$O(1)$, Medium	Realtime	Severe	$O(R)$	Low
Surge Pricing	Yes	$O(1)$, Medium	Daily & Realtime	Severe	$O(R)$	Low
Discounts & Promotions	Yes	$O(R) + O(D)$	Daily	Warning	$O(R)$	Low
Tax	Yes	$O(1)$, Small	Infrequent	Severe	$O(R)$	Low
Tolls	Yes	$O(1)$, Small	Infrequent	Severe	$O(R)$	Low
Matching Service	Yes	$O(1)$, Medium	Daily/Weekly	Warning	$O(R)$	High
Route	No	-	-	-	$O(R)$	High
Time Estimation	No	-	-	-	$O(R)$	High
Price Estimation	No	-	-	-	$O(R)$	Low

Table 1: Analysis of Marketplace Micro-services

for computing surge price based on forecast requires data analytic capabilities and access to past surge pricing and supply/demand ratios.

Matching Algorithm. The matching service is key to selecting a good match among a set of riders and set of drivers. Data scientists must constantly improve this algorithm to optimize for mission-critical goals. In order to achieve this, they need access to matches produced in the past. To support this, we log appropriate data for each matching query and the respose to and from the matching service.

Discounts and Promotions. We have information about rides taken by a rider in the past. For each rider UberLite maintains a user profile to figure out a good incentive model to encourage more rides. Similarly, for drivers, we know how much time the driver drives at what price range. Given a certain amount of money for discounts and promotions, this analysis produces a good strategy for spending it to encourage more rides overall.

6 INFRASTRUCTURE

So far, we have focused on the functionality of the core and background services within UberLite. In this section, we discuss the various types of data storage, realtime and analytics systems we would need to implement and deploy these micro-services. We would like to highlight here that background and core services services integrate together to

Service	Data Dependencies	Output	Compute
Forecasting	Historical supply/demand Special events Weather reports Realtime changes	Stored as forecasts	Complex Data Analytics Realtime
Pricing	Supply/demand forecasts Historic rider/driver declines Historic prices and supply/demand	Stored and consumed by SPS	Complex Data Analytics Realtime
Matching	Log of previous match instances	Matching Model	Complex Data Analytics
Discounts & Promotions	Historic surge prices and rider/driver declines	Stored and consumed by DPS	Complex Data Analytics

Table 2: Summary of data and compute requirements for Background Services

improve UberLite by feeding each other with required data: background services produce data that is used by mission-critical core services; core services log their every action to produce data that is then used by background services for analysis to further improve core services.

Trip Store. Trip service is the primary end-point for all trip related data. A data store serving the trip service hence must have the following properties:

- It must scale horizontally with the number of rides since a service like UberLite can grow very quickly
- The trip data maybe generated at different points in time and these pieces of info may arrive asynchronously as the people involved in the trip update the Trip Service.
- The schema must be extensible since in a fast moving startup like UberLite, we must be able to change what we collect and store very often.
- It must support a triggering service that can set hooks and listen on changes to data at a record granularity.
- Since Trip Store is the source of truth within UberLite, it must allow indexing on a large number of attributes such as rider or driver identity.

There are several open-source key-value and document stores that can be potentially modified to satisfy all the requirements listed above. However, none of them - to the best of our knowledge - support a fine-grained triggering service and allow a large number of secondary indexes. Following are some options:

- Apache HBase [14] is an open-source, distributed, versioned, non-relational database modeled after Google's Bigtable built on top of HDFS and Hadoop stack.
- Apache Accumulo [11] is a sorted distributed key/value store that is also based on Google's Big Table design, but with more advanced sercurity features.
- RocksDB [9], LevelDB [17] are two open source key/value stores based on LSM trees [20]. It is available as an embedded persistent key/value store that can be deployed as a distributed key/value store when necessary. Facebook is known to use RocksDB extensively as a distributed data storage engine.
- Project Voldemort modeled after Amazon's Dynamo [7] is a distributed data store that is designed as a key-value store used for high-scalability storage. Voldemort is a big, distributed, fault-tolerant, persistent hash table.

Trigger Framework. To our knowledge there isn't a system designed to work out-of-the-box as a trigger system. However this framework we want can be built out of Kafka thanks to the event model. Every state change can be modeled as a Kafka even topic and several consumers groups can subscribe to the appropriate event topics. However, this could serve as a scalability bottleneck. We believe that it is possible to design a tightly coupled triggered framework that can scale with the trip store by exploiting sharding within the trip store. Uber has designed their own triggering framework [27] on top of their scalable trip store.

Core Services. Core services such as Tax Service, Tolls Service, Discounts & Promotions Service require data storage as well. However, the data that powers these services are are not updated frequently and primarily serve to be used during price estimation. So, this data can be stored persistently in off-the-shelf RDBMSs with a caching layer on top to support faster reads. Some standard options are for building the persistent database layer are as follows:

- PostreSQL [22] is an open-source database that is actively developed and maintained for over 30 years.
- MySQL [30] is an alternative that is widely used by many companies such as Facebook, Twitter and YouTube.

Uber is known to have used [26] PostgreSQL internally and recently moved [25] to MySQL. Some options for building the caching layer are:

- Redis [23] is a data structure store software that can be used as a cache, message broker, and database. It is widely

used with companies such as Facebook to build a distributed caching layer on top of databases such as MySQL.
- Memcached [18] is similarly a memory object caching software. Notable users include Wikipedia, Flickr, WordPress.
- FASTER [4] [1] is an open-source embedded highly concurrent key-value store for point operations, that combines a highly cache-optimized concurrent hash index with a novel self-tuning data organization and reports state-of-the-art performance.

Realtime Services. On the other hand, services such as Driver Discovery, Traffic Monitoring, Surge Pricing have a very high update as well as read traffic and hence require a fast key/value store both for persistent storage and caching. One could use any of the key/value stores mentioned earlier to serve them. For services that involve computation on incoming streams of data/events, we have several stream processing systems such as Apache Spark Streaming [31], Apache Storm [16], Apache Flink [13], Apache Kafka [19] and Apache Pulsar [10]. Each system has specific operational characteristics that determine the scale, latency-throughput trade-off and programming flexibility. We defer their comparison as future work in the interest of space.

Logging and Analytics. The mission-critical core services do extensive logging to capture the system state regularly. This information is later used by the background services to analyze and improve the core services over time. For instance, the state of Driver Discovery Service (supply of drivers) and Surge Pricing are logged every few seconds. This log is then used in optimizing the Surge Pricing scheme to price rides appropriately. Similarly the matching service logs every input and the corresponding output produced so that the Matching algorithms team can analyze these data to improve the matching model. We may use a pub-sub system such as Apache Kafka [19] or Apache Pulsar [10] to log these services data into a distributed filesystem such as HDFS [24]. An emerging alternative is Log Device [8], which is an open-source distributed logging platform developed at Facebook.

Then logged data can be easily consumed for analytics using Hadoop [29] or Map Reduce frameworks [6]. To support complex analytics one could use Apache Spark [32] such as SparkSQL or SparkML. Other open-source tools for deep learning such as TensorFlow [2], TVM [5], PyTorch [21] can directly consume data from such distributed file systems. For SQL-like analytics, this data can be ingested into an open source distributed database such as Apache Hive [15], Apache Cassandra [12] or Myria [1] and then operated on.

[1] Disclaimer: One of the authors was involved in the design and development of FASTER

7 LESSONS LEARNED

Over the course of designing UberLite, we learnt so much about how to architect a complex service by breaking it down into manageable simpler services. Below is an attempt at summarizing these lessons succinctly:

- *Start with a small-scale monolith.* We first designed UberLite for very small scale using a single powerful database. This initial design helped bootstrap our design process, after which we focused on logically separable tasks and analyzed how each one of them behaves as we scale the operation. This strategy provided a lot of direction to the overall design process.
- *Follow the data.* Once we obtained a reasonable modular design, we analyzed the data dependencies within and across each module. Then the natural question that followed was how do we enable this data dependency within the system? Answering this helped us quickly come up with a physical design from functional specification.
- *Design for flexibility.* At every stage of the design, we prioritized flexibility since our end goal was also to extend UberLite for ride-sharing. While premature optimization could be an evil to be avoided, designing for the future both in terms of new features and performance growth is key to achieving a good extensible system design.
- *Open-source is powerful.* As we started investigating potential open-source systems that could be used to serve our functional needs, we realized how the open-source ecosystem could satisfy almost every single requirement barring a few exceptional ones. Getting a complex application such as UberLite up and running can be done in a matter of few months with the help of mainly open-source systems and this is a great thing to be true!

REFERENCES

[1] 2017. *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2017/index.html

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. http://tensorflow.org/ Software available from tensorflow.org.

[3] CBInsights. 2019. *How Uber Makes — And Loses — Money.* https://eng.uber.com/mezzanine-migration/

[4] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA, 275–290.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-end Optimizing Compiler for Deep Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, Berkeley, CA, USA, 579–594. http://dl.acm.org/citation.cfm?id=3291168.3291211

[6] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6 (OSDI'04)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=1251254.1251264

[7] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. ACM, New York, NY, USA, 205–220. https://doi.org/10.1145/1294261.1294281

[8] Facebook. 2019. *LogDevice*. https://logdevice.io/

[9] Facebook. 2019. *RocksDB*. https://rocksdb.org/

[10] The Apache Software Foundation. 2016. Apache Samza. http://gopulsar.io/. Accessed: 2017-01-16.

[11] The Apache Software Foundation. 2019. *Apache Accumulo*. https://accumulo.apache.org/

[12] The Apache Software Foundation. 2019. *Apache Cassandra*. https://cassandra.apache.org/

[13] The Apache Software Foundation. 2019. *Apache Flink*. https://flink.apache.org/

[14] The Apache Software Foundation. 2019. *Apache HBase*. https://hbase.apache.org/

[15] The Apache Software Foundation. 2019. *Apache Hive*. https://hive.apache.org/

[16] The Apache Software Foundation. 2019. *Apache Storm*. https://storm.apache.org/

[17] Google. 2019. *LevelDB*. https://github.com/google/leveldb

[18] Memcached. 2017. https://memcached.org/. [Online; accessed 30-Oct-2017].

[19] Neha Narkhede, Gwen Shapira, and Todd Palino. 2017. *Kafka: The Definitive Guide Real-Time Data and Stream Processing at Scale* (1st ed.). O'Reilly Media, Inc.

[20] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-structured Merge-tree (LSM-tree). *Acta Inf.* 33, 4 (June 1996), 351–385. https://doi.org/10.1007/s002360050048

[21] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[22] R Development Core Team. 2004. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria. http://www.R-project.org 3-900051-07-0.

[23] Redis. 2017. https://redis.io/. [Online; accessed 30-Oct-2017].

[24] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (MSST '10)*. IEEE Computer Society, Washington, DC, USA, 1–10. https://doi.org/10.1109/MSST.2010.5496972

[25] Uber. 2016. *The Architecture of Schemaless, Uber Engineering's Trip Datastore Using MySQL*. https://eng.uber.com/schemaless-part-two/

[26] Uber. 2016. *Project Mezzanine: The Great Migration*. https://eng.uber.com/mezzanine-migration/

[27] Uber. 2016. *Using Triggers On Schemaless, Uber Engineering's Datastore Using MySQL*. https://eng.uber.com/schemaless-part-three/

[28] Uber. 2018. *H3: Uber's Hexagonal Hierarchical Spatial Index*. https://eng.uber.com/h3/

[29] Tom White. 2009. *Hadoop: The Definitive Guide* (1st ed.). O'Reilly Media, Inc.

[30] Michael Widenius and Davis Axmark. 2002. *Mysql Reference Manual* (1st ed.). O'Reilly & Associates, Inc., Sebastopol, CA, USA.

[31] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. 2012. Discretized Streams: An Efficient and Fault-tolerant Model for Stream Processing on Large Clusters. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing (HotCloud'12)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=2342763.2342773

[32] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65. https://doi.org/10.1145/2934664