The Modern Data Architecture

The Deconstructed Database

AMANDEEP KHURANA AND JULIEN LE DEM

Amandeep Khurana is cofounder and CEO at Okera, a company focused on solving data management challenges in modern data platforms. Previously, he was a Principal Architect at Cloudera where he supported customer initiatives and oversaw some of the industry's largest big data implementations. Prior to that, he was at AWS on the Elastic MapReduce engineering team. Amandeep is passionate about distributed systems, big data, and everything cloud. Amandeep is also the coauthor of *HBase in Action*, a book on building applications with HBase. Amandeep holds an MS in computer science from the University of California, Santa Cruz. amansk@gmail.com

Julien Le Dem is the coauthor of Apache Parquet and the PMC chair of the project. He is also a committer and PMC Member on Apache Pig, Apache Arrow, and a few others. Julien is a Principal Engineer at WeWork working on data platform, and was previously Architect at Dremio and Tech Lead for Twitter's data processing tools, where he also obtained a two-character Twitter handle (@J_). Prior to Twitter, Julien was a Principal Engineer and Tech Lead working on content platforms at Yahoo, where he received his Hadoop initiation. His French accent makes his talks particularly attractive. julien@ledem.net

Mainframes evolved into the relational database in the 1970s with the core tenet of providing users with an easier-to-use abstraction, an expressive query language, and a vertically integrated system. With the explosion of data in the early 2000s, we created the big data stack and decoupled storage from compute. Since then the community has gone on to build the modern data platform that looks like a deconstructed database. We survey the different technologies that have been built to support big data and what a modern data platform looks like, especially in the era of the cloud.

Modern data platform architectures are spurring a wave of innovation and intelligence by enabling new workloads that weren't possible before. We will review three main phases of technology evolution to highlight how the user experience of working with data has changed over time. The article concludes with a review of the current state of data architectures and how they are changing to better meet demand.

From Mainframe to Database—A Brief Review

Mainframes were among the early platforms for applications and analytics done in a programmatic way, using what we know as modern computing systems. In the world of mainframes, users had to write code to interact with data structures as stored on disk. Users had to know the details of the data storage with which they were working, including its location and storage format. These details had to be coded as a part of the application. You could still write arbitrarily complex logic to work with the data, but the paradigm was not very accessible or easy to understand by mainstream users. The technical complexity was a hurdle users had to overcome, thus limiting the adoption of this paradigm.

Fortunately, in the 1970s, the relational database was born. It was created based on a few core tenets:

- Simplify the data abstraction for the end user and make it more intuitive.
- Provide a rich language to facilitate the expression of computational logic.
- Hide the complexity of the underlying systems from end users.

These goals are clearly articulated in the first paragraph of Codd's 1970 paper on relational models [1], one of the first papers on relational databases. You don't have to read much past the first three sentences of his paper:

> *Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory solution. Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed. Changes in data representation will often be needed as a result of changes in query, update, and report traffic and natural growth in the types of stored information.*

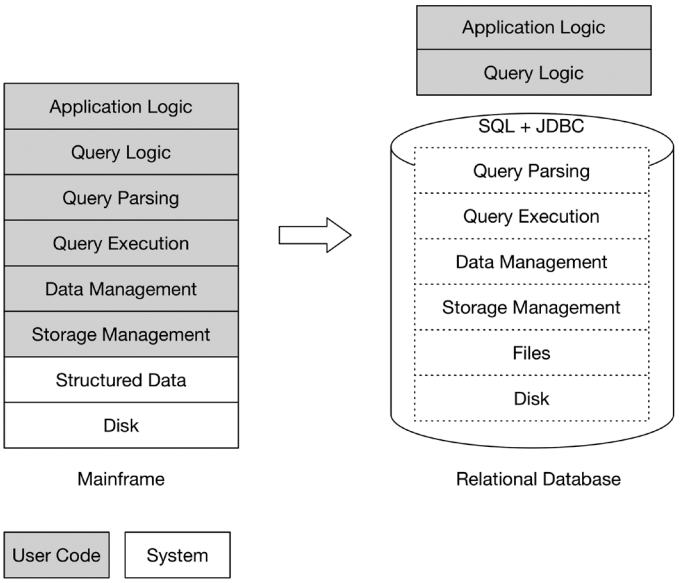

Figure 1: The evolution of the technology stack from the mainframe to the database

This means:

- Users of database systems should not have to worry about the underlying layouts, how and where data is stored, formats, etc.
- If changes need to be made to underlying files and structures, the applications should not be affected.
- Anything that provides more and better information about the underlying data structure doesn't necessarily reduce the technical complexity.

One could argue that data analytics, as we know it today, was made possible by the relational database. For the first time, companies could leverage their data and extract value from it. The relational database employed SQL (created in the early 1970s at IBM) as the language to express computation, and 1986 saw the first SQL standard, which has been updated several times since, eventually becoming a global standard. SQL has some excellent properties, including a strong separation of the query logic from the execution details. The table abstraction and how it is stored or indexed is opaque to the user, who can concentrate on the data logic rather than the storage implementation. An optimizer is charged with finding the best way to produce the requested data and exploit the properties of the underlying storage (column-oriented, indexed, sorted, partitioned). Additionally, ACID guarantees, integrity constraints, and transactions help to ensure certain properties of the data.

In addition to a standard language to express data-processing logic, Sun Microsystems released JDBC as a standard API, which further abstracted the underlying SQL implementation from the user (see Figure 1). An entire ecosystem of technologies and applications was created around the relational database.

At the very core, it was ease of use, the accessibility and the simplicity of the database, that led to its broad adoption. You no longer needed to be an engineer to work with data.

The Birth of the Big Data Stack

In late 1990s and early 2000s, the relational database struggled to keep up with the explosion of data. Technologists who worked with large amounts of data re-evaluated data platform architectures. The reasons for this included scalability limitations, the increasingly heterogeneous nature of data, and the types of workloads people wanted to run. The database's architecture constrained these capabilities. SQL, as a language, was not expressive enough, and the database wasn't flexible and scalable enough to support different workloads.

This reconsideration of data storage and processing was the genesis of the big data stack and, later on, the concept of the data-lake. The Apache Hadoop project was at the core of this. Hadoop started in 2006 as a spin-off from Apache Nutch, a web crawler that stemmed from Apache Lucene, the famous open source search engine. The inspiration for this project came from two Google papers describing the Google File System [2] and a distributed processing framework called MapReduce [3]. These two components combined the extreme flexibility and scalability necessary to develop distributed batch applications in a simple way.

The Hadoop Distributed File System (HDFS)

HDFS provides a file system abstraction over a cluster of mainstream servers. It also provides metadata on data placement, which is exploited by MapReduce to process data where it is stored. Back when network I/O was much more constrained than disk I/O, this innovation was significant. HDFS files are free form; there are no constraints on the format or any kind of schema. We started to call this concept *schema on read* (as opposed to *schema on write* in the world of databases).

MapReduce

MapReduce provides a simple framework to build distributed batch applications on top of HDFS. Usually a job is defined by scanning a data set in parallel, applying a Map function to the content, and emitting key-value pairs. All values with the same key are sent to the same machine, independent of where they were produced, in a step called "the shuffle." The key and its corresponding list of values are then passed to the Reduce function. This simple framework allows us to build powerful distributed algorithms. One example is the famous PageRank algorithm, originally used by Google to rank websites based on how many incoming links they get.

MapReduce is a very flexible paradigm. MapReduce simply edits key-value pairs, and it is also composable, allowing its users to

realize complex algorithms by orchestrating multiple MapReduce steps. For example, PageRank converges to a result after a number of iterations. The inherent limitations of MapReduce, however, come from the same attributes that make it strong. The flexibility in file formats and the code used to process them offer no support for optimizing data access. In that respect, the MapReduce paradigm returns us to the world of mainframes at a much larger scale. The MapReduce programmer must in fact know quite a bit about storage details to write a successful program.

"MapReduce, a Major Step Backwards"

The database community eventually became annoyed by this new wave of open source people re-inventing the wheel. Turing Award winner Michael Stonebraker, of PostgreSQL fame and recent co-founder of the distributed analytical database Vertica, famously declared in 2008 [4] that MapReduce was "a major step backwards." Compared to the nice abstractions of the relational model, this new model was too low level and complex.

Evolution of the Big Data Stack

Ten years later, the entire Hadoop ecosystem is much larger than the two components it originally included. People argued about where the boundary of that ecosystem really stopped. In the cloud, you can even use a significant portion of the ecosystem without Hadoop itself. New functional categories beyond storage and compute have emerged: execution engines, streaming ingest, resource management, and, of course, a long list of SQL-on-Hadoop distributed query engines: Impala, Hive, SparkSQL, Drill, Phoenix, Presto, Tajo, Kylin, etc. The ecosystem can be broken down into the following categories:

Storage Systems

HDFS, S3, and Google Cloud Storage are the distributed file system/object stores where data of all kinds can be stored. Apache Parquet has become a standard file format for immutable columnar storage at rest.

Apache Kudu and Apache HBase provide mutable storage layers with similar abstractions, enabling projection and predicate pushdown to minimize I/O by retrieving only the data needed from disk. These projects require explicit schema, and getting that right is critical to efficient access.

Streaming Systems

Kafka is the most popular stream persistence system in the data platform world today. It's open source and is widely used for streaming data at scale. Kinesis, an AWS service, is the most popular hosted and managed streaming framework but is not available outside the AWS environment. GCP provides a similar service called PubSub. Another noteworthy platform is Pulsar. Pulsar has interesting features for multi-tenancy and performance of concurrent consumers on the same stream. The project is more of a challenger that has yet to reach wide adoption.

Query Engines

Since MapReduce, many other query engines have developed. Originally, they were often layered on top of MapReduce, which introduced a lot of latency; MapReduce is designed for web-scale indexing and is optimized for fault tolerance, not quick response. Its goal is to optimize for running very large, long-running jobs, during which a physical failure of at least one component is likely.

For example, Hive (an SQL implementation) and Pig (a functional DSL with similar capabilities) both originally compiled to a sequence of MapReduce jobs.

As more people wanted interactive capability for data analytics, Hadoop-compatible data-processing engines evolved and MapReduce became less relevant. Spark has become a popular alternative, with its richer set of primitives that can be combined to form Data Availability Groups (DAGs) of operators. It includes an in-memory cache feature that allows fast iterations on a data set during a session of work. Tez is a lower level DAG of operator APIs aimed at optimizing similar types of work. SparkSQL is a SQL implementation on top of Spark. Hive and Pig both can now run on either MapReduce, Spark, or Tez.

There are several SQL engines that provide their own runtime, all with the goal of minimizing query latency. Apache Drill and Apache Presto generate optimized Java bytecode for their operators, while Apache Impala uses LLVM to generate native code. Apache Phoenix supports a SQL interface to Apache HBase and takes advantage of HBase's capabilities to reduce I/O by running code in the database.

Tools such as Python, with the help of libraries like NumPy and pandas and R, are also very popular for data processing and can cater to a large variety of use cases that don't need the scale that MapReduce or Spark supports.

In addition to these, we're seeing a variety of machine-learning frameworks being created, such as Tensorflow, DL4J, Spark MLLib, and H2O. Each of these is specialized for certain workloads, so we are going to see more of these emerge over the next few years.

Query Optimizer

There is a query parser and optimizer framework in the Calcite project, one of the lesser known but most important projects in the ecosystem. Calcite powers the query execution in projects such as Hive, Drill, Phoenix, and Kylin. Calcite is also used in several streaming SQL engines such as Apex, Flink, SamzaSQL, and StormSQL. It can be customized in multiple ways. Notably, one would provide an execution engine, schema, and connectors implementations as well as optimization rules either relevant to

the execution engine, the storage layer, or both. Spark, Impala, and Presto have their own query optimizers and don't use an external engine.

Serialization

Apache Arrow is a standard in-memory columnar representation that combines efficient in-memory query evaluation, allowing for zero-overhead serialization, with standard simplifying integration, removing unnecessary and costly conversion layers. It also allows fast in-memory processing by enabling vectorized execution.

Security

Access control policies can be put in policy stores like Apache Sentry and Apache Ranger. In addition, there are proprietary tools such as BlueTalon that enable access control on SQL engines.

Cataloging and Governance

There are a few offerings in this realm as well, but none that truly solve the problems of today. The Hive Metastore is the dominant schema registry. Apache Atlas is an open source framework that's focused on governance. Proprietary tools such as AWS Glue, Cloudera Navigator, Alation, Waterline, and Collibra are solving different aspects of the problem.

Towards a Modern Data Platform—The Deconstructed Database

In parallel to the evolution of the data-lake concept and the big-data stack, the world of cloud computing continues to redefine technology architectures. Cloud computing normalizes variants of infrastructure, platform, and applications as a service. We are now seeing the emergence of Data-as-a-Service (DaaS). All these trends constitute a significant paradigm shift from the world of datacenters, in which enterprises had to either build their own datacenters or buy capacity from a provider. The kind of data platform that people want to build today, especially in the cloud, looks very different from what we have seen so far. At the same time, the modern data platform borrows many of the core tenets we valued in previous generations. The core tenets of the cloud include:

1. Allowing **choice** between multiple analytics frameworks for data consumers so they can pick the best tool for the workload.
2. **Flexibility** in the underlying data source systems but a consistent means to enable and govern new workloads and users.
3. A **self-service experience** for the end user: no waiting on IT and engineering teams to catch up and deliver on all the asks from all the constituents they have to serve.

Agility and self-service require components to be loosely coupled, easily available as a service or open source software, and usable in different contexts. Systems that are loosely coupled need to have common, standard abstractions in order to work together. Many of these are missing today, which makes building a true modern data platform with the core tenets articulated above challenging.

Given where the ecosystem is headed, new developments are enabling the capabilities that people want. Key areas that are experiencing significant innovation include:

1. **Improved metadata repository and better table abstractions**. There are many promising projects maturing in the open source ecosystem. For example, the Iceberg project from Netflix defines table abstractions to provide snapshot isolation and serialization semantics (at a high level, not row by row) to update data in a distributed file system. Iceberg abstracts away formats and file layouts while enabling predicate and projection push downs. Marquez is also a project that defines a metadata repository to take advantage of this work.
2. **Access control and governance across different engines and storage systems**. Current methodologies are fragmented and not fully featured. In the wake of GDPR and other privacy acts, security and privacy are important aspects of the data platform. Labeling private data appropriately to track its use across the entire platform, and enabling only approved use cases, has become a key requirement. The current ecosystem does not deliver on this, and there are new developments that will take place to fill this gap.
3. **Unifying push-down logic**. A great help toward more consistent performance of query engines on top of Parquet and other columnar storage would be unifying push-down logic. Current implementations are very fragmented and duplicate effort. The same concepts apply to streaming.
4. **Arrow project adoption to enable better interoperability between components**. This would enable simpler and more general interoperability between systems but, more importantly, would do so without sacrificing performance as lowest common denominator type integrations often do.
5. **A common access layer**. A unified access layer that allows push-downs (projection, predicate, aggregation) and retrieves the data in a standard format efficiently will advance modern data architectures. We need this capability whether or not data storage is mutable (HBase, Cassandra, Kudu), batch-oriented (HDFS, S3), or streaming-oriented (Kafka, Kinesis). This unified access layer will improve interoperability and performance, reduce duplication, and support more consistent behavior across engines. A lot of other data management problems can be solved at this layer. This is also in line with Codd's core tenet of databases: users of large data banks should not have to deal with the internal semantics of data storage.

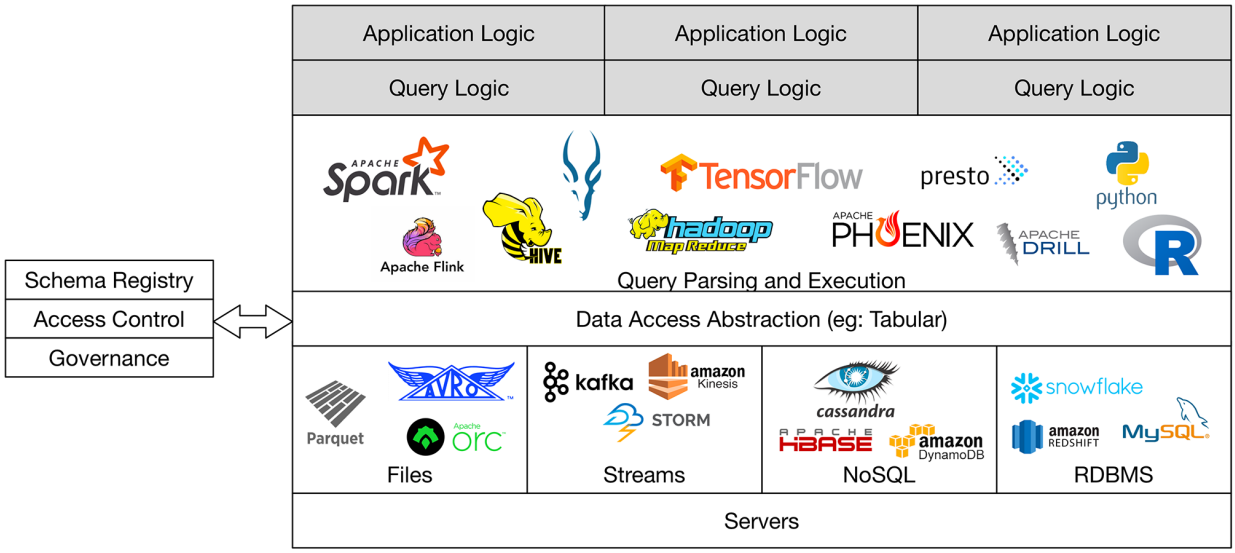

Figure 2: The modern data platform stack

Conclusion

Bringing it all together, a modern data platform will look similar to the stack shown in Figure 2. It will have a subset of these components integrated as independent, specialized services. The figure shows a few examples of the technologies at various levels of the stack and is not an exhaustive list.

A typical deployment may not always consist of all the components shown. Platform owners will be able to pick and choose the most appropriate ones and create their own stack, giving end users the flexibility and scale they need to run new workloads in the enterprise. This modular approach is a powerful paradigm that will further enable new capabilities for enterprises. This will drive more innovation and disruption in the industry, making businesses data-driven by shortening time to market of applications that take advantage of the large volumes of data that are defining the modern enterprise.

References

[1] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, vol. 13, no. 6 (June 1970), pp. 377–387: https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf.

[2] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*: https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf.

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1 (January 2008), pp. 107–113: https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf.

[4] D. J. DeWitt and M. Stonebraker, "MapReduce: A Major Step Backwards," The Database Column, 2008: http://db.cs.berkeley.edu/cs286/papers/backwards-vertica2008.pdf.