How to Design Disaggregation in Large-Scale Transaction Systems

Zhihan Guo
University of Wisconsin-Madison

Nowadays, organizations have increasing desire for a transaction system that provides flexible, scalable, and highly available services across geo-distributed data centers with ACID guarantees. Many databases are built on this purpose including but not limited to Amazon Aurora [5], Azure SQL DB Hyperscale (Socrates) [1], CockroachDB [4], FoundationDB [6], Google Spanner [3], PolarDB [2]. These database systems feature a disaggregated architecture that decouple different components as multiple distinct subsystems. Most of them decouple the storage from the computation as a standalone service, as an example. This allows each part to scale and bill independently while improving resource utilization, reducing operational cost, and enabling flexible cloud deployment with heterogeneous configurations. However, the designs vary greatly across different systems — each system has a distinct architecture and a specific set of techniques. Our goal here is to understand the design space.

Disaggregation Design Space. The first question is what can be disaggregated and what is the spectrum of the possible physical arrangements. For example, on one hand, CockroachDB has two main physical clusters of machines – one for computation-related tasks such as SQL parsing and execution while the other cluster takes charge of the storage and handles read/write requests. Storage nodes handle both conflict detection and fault tolerance. On the other hand, FoundationDB breaks down the system into more standalone subsystems connecting through the network such as metadata management (the proxies), concurrency control, timestamp generation, logging (the durable queuing system), and table storage.

What to disaggregate. The next question is whether to disaggregate a component. There are many factors leading people to choose disaggregation such as different scalability requirements, privacy and isolation concerns, heterogenous hardwares, deployment of existing cloud services, etc. For instance, Google Spanner uses Colossus to exploit the existing durability and scalability provided by an existing distributed file system. Although there are many reasons for disaggregation, the cost in performance and resources is also worth considering. As an example, people may decouple the metadata storage from the main data storage for considerations such as privacy and isolation. However, this separation need to deploy two subsystems to provide similar consistency and availability guarantees. As a result, it may not only take more resources and development efforts, but also increase the latency — in a case that a data access request queries both subsystems for the metadata such as locks and partition information and the table data.

Disaggregation Challenge 1: Correctness. Moreover, there are new issues we need to take care of due to disaggregation. One common problem is how to read the latest data in a multi-versioned database. Traditionally, fetching data and updating data go to the same machine. In a disaggregated setup, nevertheless, a client may read data from a place that is different from where the data was updated at the commit time. For one instance, FoundationDB commits a transaction when the write-ahead-logs are durable and replicated. A separate table storage will asynchoronously pull the log and apply the changes to the tables. In this case, there could be a lag so that the latest version is not yet available in the table storage. The table storage should be able to identify the case when a request arrives. Otherwise, it will provide the latest available version that is actually staled. There are other examples of data read from a copy that is not syncrhonously updated during commit such as reading from a client-side cache, a node serving read-only requests, a storage asyncrhonously materializing logs. Some systems [3, 4] avoid the issue by unifying the read/write destination while some [5, 6, 2] uses a watermark to track. Although it may firstly seem to be a tiny element that every system develops their owns as part of their integrated design, we can actually compare across the alternative plans across systems to evaluate the involved trade-offs.

Disaggregation Challenge 2: Performance. In addition to solve new problems, we also need to re-evaluate some existing solutions as the trade-off may change in the new architecture making them sub-optimal. One example is where to keep the groundtruth of metadata such as key-partition mapping. When the storage and compute is not separated, the metadata can be partitioned and kept along with the storage. As storage becomes disaggregated, keeping the mapping in storage will incur more network messages to route the request, while keeping the mapping in the compute may require extra support for strong consistency and high availability. To this end, existing systems usually takes the former approach and allow caches in the compute side, however, cache invalidation becomes a new problem especially during ongoing changes of metadata. Similarly, where to evaluate transaction conflicts also involves new trade-offs between extra network messages and complicating storage systems.

In summary, this project aims to explore the design space of disaggregation in modern large-scale transaction systems by considering the four proposed aspects and we hope to provide some useful insights when developing such a disaggregated architecture.

1. REFERENCES

[1] P. Antonopoulos, A. Budovski, C. Diaconu, A. Hernandez Saenz, J. Hu, H. Kodavalla, D. Kossmann, S. Lingam, U. F. Minhas, N. Prakash, et al. Socrates: The new sql server in the cloud. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1743–1756, 2019.

[2] W. Cao, Y. Zhang, X. Yang, F. Li, S. Wang, Q. Hu, X. Cheng, Z. Chen, Z. Liu, J. Fang, B. Wang, Y. Wang, H. Sun, Z. Yang, Z. Cheng, S. Chen, J. Wu, W. Hu, J. Zhao, Y. Gao, S. Cai, Y. Zhang, and J. Tong. *PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers*, page 24772489. Association for Computing Machinery, New York, NY, USA, 2021.

[3] D. Malkhi and J.-P. Martin. Spanner's concurrency control. *ACM SIGACT News*, 44(3):73–77, 2013.

[4] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 14931509, New York, NY, USA, 2020. Association for Computing Machinery.

[5] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052, 2017.

[6] J. Zhou, M. Xu, A. Shraer, B. Namasivayam, A. Miller, E. Tschannen, S. Atherton, A. J. Beamon, R. Sears, J. Leach, et al. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2653–2666, 2021.