This week’s material was my first real introduction to the history and background around BASE style architectures, and it was extremely informative. From my own experience, I have only worked within ACID architectures and am most comfortable within a relational database. When first starting out in my current role, our data was stored in an Oracle database, and as noted in the lectures, this is a very rigid ACID architecture where tables are “locked” whenever an INSERT, UPDATE, etc. transaction is ongoing. Recently, my organization has moved away from Oracle and our data is now housed in AWS S3 and query-able through Athena (with DataGrip being the preferred IDE). Similarly, Athena/S3 is ACID in that the data files are stored in S3 and are final to some extent, with Athena only serving as a query tool. A user can create their own tables, but the underlying data is static. The expectation is that multiple users would receive the same answer if the same query was run at a given time.

An ACID infrastructure is critical in my organization, as we work in the healthcare space and report on quality and cost of care for various client initiatives. A delay in a data copy receiving the most up to date value would be a huge issue in our line of work. We need to always have confidence that we are reporting on the most up to date information as the raw and underlying data is always a source of truth that we fall back on. An added benefit of ACID architecture when compared to a Base or NoSQL architecture is the availability to craft “the perfect SQL statement”. In our work, we’re often asked very specific questions around certain sub-population or areas of care, and it’s critical to have a very refined SQL interface that allows for these investigations.

However, while I am most familiar and comfortable with an ACID architecture, this week’s discussion taught me about the other types of approaches and the benefits that these bring. A big advantage in NoSQL and BASE-like architectures is the readily available presence of read replicas. In my mind, this would be beneficial in my own company to reduce load on the primary database when there are time sensitive deliverables at risk. Our company has multiple clients and tables that may often have billions of rows, so re-directing less time sensitive work to other locations would be extremely valuable. Additionally, while I am most comfortable in a relational database, a table structure is restrictive in some regards. The NoSQL approach of not relying on a table and instead other data structures (JSON, key:value, etc.) I believe would allow for much more flexibility in intaking data from disparate sources while still providing a universal way to query. Therefore, while in my own work I will continue to still utilize ACID, this week has been great in opening my eyes to alternates that may become more and more prevalent over time.

This reflection is to compare, contrast, and explore **ACID** and **BASE** database style architectures. More specifically, we will review ACID database **design principle**, then do the same for BASE style architecture; and we will highlight **core differences** in a short discussion.

ACID is an acronym for Atomicity, Consistency, Isolation and Durability. **Atomicity** enforces the idea that given a transaction, either all the steps within the given transaction completes or none of them completes at all. There is no half-executed transaction, but transactions within an ACID architecture are all or nothing. This preserves completeness of the whole process. The second term of the ACID acronym is **consistency**, which refers to that integrity of the data should be maintained before and after a transaction is executed. The state of data in the beginning and at the completion of transactions is expected to reflect that only valid data is saved in the process. Another characteristic of ACID style architecture is **isolation**, where any running transaction has the illusion to have no concurrent transaction. To ensure that transactions do not affect each other, a locking mechanism is implemented so that partially intermediate value are not available to other concurrent modifications.  The system is running only a single transaction at the time. Finally, **durability** refers to the impact of an outage or a failure on a running transaction. Once the commit happens, it is irreversible meaning written data cannot be lost. So a durable transaction will not impact the state of (written) data if the transaction ends abnormally, in other words the data survive any sort of failures and the data integrity will be maintained.

An alternative to ACID approach is BASE architecture, which acronym stands for Basically Available, Software, Eventual consistency. **Basically Available** means the system responds to any request; there is no guarantee of any piece of data to be available but this doesn’t halt the system. The lack of guarantee is explained by the **soft state**, i.e. an intermediate state that will eventually change over time. System changes are constantly happening but the data you retrieve at a given point of time may eventually be overwritten by more recent data. Now it is in an intermediate state and if someone calls and requests, he will get the intermediate value. But eventually the soft state will move to a consistent state; **eventually consistent** is the guarantee of consistency over time nonetheless there are times when the database is in inconsistent state. For example, when multiple copies of the data reside in separate servers which is database replication, or a fundamental feature in cloud where you have multiple copies of your data to support high availability, failure, etc. Updates may not be immediately made in all copies simultaneously; so the data is inconsistent for a period of time but the replication mechanism will eventually update all of the copies of the data to be consistent. The focus here is availability, thus consistency will be compromised because we want the system to be available all the time. When you make changes it takes time for changes to be reflected in other servers, so during that time the server is in a soft state; and once updates get reflected, eventually all servers move to consistent state – that’s BASE transaction.

From a comparison perspective, focusing on availability compromises consistency and vice versa. An ACID system ensures in a 2-phase commit that data distributed across servers are consistent before released for subsequent queries. Yet immediate consistency across distributed partitions (servers) limits scale-out performance. On the contrary, BASE focuses on availability and not on consistency. So eventual consistency is acceptable in BASE architectures so it’s not necessary to hold subsequent queries until updates are fully written to distributed partitions. As a result, scale-out performance is greatly enhanced. This is fine when the nature of the data can tolerate some imprecision in query results. In short it is important to keep in mind what ACID means and what data integration means for your applications and business requirements. Only by being informed and implemented the right technology for your need you will be able to succeed for your database and data management systems. For example, medical records or banking transactions require consistency so ACID design is preferred. On the contrary, BASE architecture works well for a sentiment analysis on tweets changing over time since twitter feeds is not structured, data scope is not bounded and doesn’t necessarily need data consistency.