All Articles

Why the PIE theorem is more relevant than the CAP theorem

In this post, you'll learn:

- Why the CAP theorem is irrelevant in most data choices
- What the PIE theorem is and why it matters
- The three PIE theorem options
- How serverless compute complicates the picture

The CAP theorem is probably the most popular theorem in computer science. At a very simplified level, failures will happen in a distributed system. When a failure does happen, the CAP theorem says you will need to choose between *consistency* (clients always read the most recent data) and *availability* (clients always get a non-error response).

It's fun to dream about building large-scale distributed systems and how you'll handle the difficult choice of availability vs. consistency. Of course you'll choose the cutting-edge database that avoids the CAP theorem entirely (and heck, maybe it solves the P vs NP problem while we're at it)!

Joking aside, I'm going to tell you a little secret — you shouldn't really care about the CAP theorem.

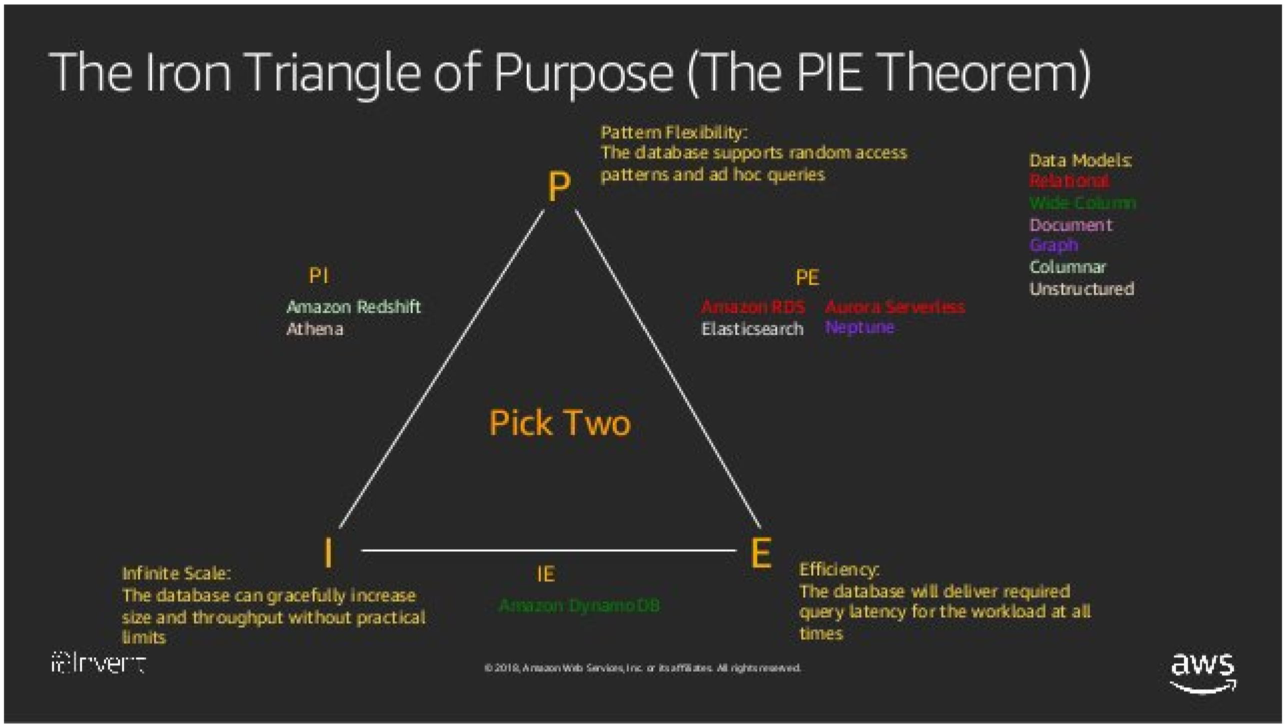
Here's some advice that works for 95% of you: **choose consistency over availability**. It's good enough for Google. It's easier to reason about. And you probably don't have such fickle users that a small bit of downtime will make or break you.

You shouldn't really care about the CAP theorem.

But there is another theorem you should know about when making database decisions. And this one is much tastier.

Introducing: The PIE Theorem.

Rick Houlihan, the breakout star of AWS re:Invent, had a session on [choosing the right database for your workload](). In the talk, he introduced a concept called the PIE theorem:


The Iron Triangle of Purpose (The PIE Theorem)
Pattern Flexibility:
The database supports random access patterns and ad hoc queries
P
Data Models:
Relational
Wide Column
Document
Graph
Columnar
Unstructured
PI
Amazon Redshift
Athena
PE
Amazon RDS
Aurora Serverless
Elasticsearch
Neptune
Pick Two
I
E
Infinite Scale:
The database can gracefully increase size and throughput without practical limits
IE
Amazon DynamoDB
Efficiency:
The database will deliver required query latency for the workload at all times
re:Invent
© 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.
aws

The PIE theorem posits that you can choose two out of three desirable features in a data system:

- **Pattern Flexibility**
- **Efficiency**
- **Infinite Scale**

Let's dive a little deeper into what each of these means.

Pattern Flexibility asks the question 'Do you want the option to ask new, unanticipated questions of your data?'

For example, maybe you're writing a bookstore and originally modeled your books to be queried by ISBN code. A database with pattern flexibility would allow you to easily access by author name in the future, as your requirements change.

Efficiency asks about the basic performance characteristics of your database.

Do you need to serve many distinct queries at the same time with response times of under a few hundred milliseconds? If so, efficiency ought to be one of your two choices. If you only need a few concurrent queries and can tolerate latency of seconds or even hours, then efficiency is not a primary concern.

Infinite scale asks 'how big do you expect your data to grow?'

More data means more other resources — disk, compute, network, etc. Are you building the next Snapchat, with hundreds of millions of users? Or are you building an application that won't reach millions of users or hundreds of GB of data?

The era of purpose-built datastores is here. The next time you have a project that considers a data store, the PIE theorem should be top of mind. The examples below show how the three permutations of the PIE Theorem (PE, PI, and IE) play out.

The Traditional Choice: Pattern Flexibility & Efficency

The default choice for many years has been PE — pattern flexibility and efficiency.

For line-of-business applications and other similar projects, PE is a perfect choice. These applications require reasonably fast reads and writes from multiple clients at the same time, so efficiency is a must.

Further, query flexibility is important. These applications often evolve over time as users ask for additional use cases. Also, you may use the same database both for serving on-line queries and for running analytic queries. The flexibility of a PE data store is helpful here.

Relational databases are the most popular option in this category. All the time you spent learning about database normalization and third normal form is for pattern flexibility. It's simple to accommodate new use cases or ask ad-hoc questions with a normalized data model. Relational databases are also highly-efficient as long as your hardware is sufficient and your queries aren't too complex.

This is the most popular category by a longshot — 7 of the top 10 databases in the db-engines.com rankings are relational databases.

However, PE data stores don't solve every need. In the next section, we'll learn about the emergence of IE data stores.

Web Scale with NoSQL: Infinite Scale & Efficiency

The growth of the internet gave rise to a new problem — immense scale. Companies like Amazon, Facebook, and Google have billions of users, all demanding data as quickly as possible.

Under this scale, the PE systems began to break down. You could use bigger and bigger hardware for your relational database but even Moore's law couldn't keep up.

This increased scale identified two pain points with the PE regime:

1. It was impossible or infeasible to host all data on a single machine; and

2. Relational JOINs proved inefficient at scale due to the additional I/O and computation required.

In response to this, we saw the rise of NoSQL. NoSQL systems eschewed the normalized nature of their relational predecessors. Rather than optimizing for write simplicity and read flexibility, NoSQL emphasized read simplicity. Your data should be organized to allow for simple, fast lookups to specific use cases.

This denormalized nature also made it easier to horizontally shard your data. Previously, spreading subsets of the data across multiple machines could be expensive when making JOINs across machines, as network I/O is significantly slower than hitting local memory or disk. However, if your data is pre-assembled in the format needed by the client without any JOINs, each query can hit the specific node that contains the relevant data without any inter-node communication.

Here, we see the trade of **pattern flexibility** for **infinite scale**. With a denormalized data store, you lose the flexibility you had from normalization. It's possible that a new access pattern will require a rewrite of your data. However, by being specific with your access patterns, you have made it easy to scale your data store to billions of users with consistent, predictable latency.

The most common IE systems, according to the db-engines rankings, are MongoDB (#5), Cassandra (#11), HBase (#17), and DynamoDB (#20).

In the next section, we'll cover the final PIE permutation — PI systems.

The Data Warehouse: PI

The last choice for the PIE theorem is the PI system. In this system, a user wants **pattern flexibility** and **infinite scale**. However, they're willing to give up **efficiency**, in the sense that they don't need sub-second response times or thousands-to-millions of concurrent queries.

The canonical example here is the data warehouse. I think it's useful to think of this in conjunction with the IE systems discussed previously. In the move from relational systems (PE) to NoSQL systems due to scale, internal users lost the ability to run ad-hoc queries over their data. Copying your data from an IE system into a PI system gives you back the flexibility you want for analytic queries.

The rise of the data warehouse is more than just about NoSQL though. These powerful PI systems allow you to combine multiple data sources, including production databases, event logs, and SaaS metrics, into a single source of truth.

The earliest versions of PI systems are tools like Teradata, SAP HANA, or Vertica. More recently, Amazon Redshift and Snowflake have exploded in popularity.

We've been talking about full database management systems throughout this piece. However, I'd also put progeny of the Google MapReduce paper in the PI bucket. Tools like Hadoop MapReduce or Apache Spark allow for very flexible access patterns on datasets of infinite scale — you just pay a high price for efficiency as you need to wait for your application to execute.

The Serverless Complication

I'm a serverless believer, so it wouldn't be right not to discuss one more aspect which can complicate your database decision — the rise of serverless compute.

At a first glance, serverless compute shouldn't affect your database choice at all — whether you're using bare metal servers in your own datacenter, Docker containers on Google Kubernetes Engine, or serverless functions on AWS Lambda, the PIE theorem is about what scale and access properties you need out of your database.

However, there is still some difficulty in using serverless compute with existing databases. Namely, most databases weren't made for a world of hyper-ephemeral compute.

Most databases weren't designed for a world of hyper-ephemeral compute.

The most common place you see this pop up is in database connections. Traditional relational systems often impose connection limits (see these links for background on MySQL connection limits and Postgres connection limits).

Relational systems were built for a world of long-running compute. These compute instances would initialize connection pools to the relational databases and reuse these connections across requests. In the world of hyper-ephemeral serverless compute, this doesn't work. Each function instance is logically, and often physically, separate from the others, so connection pooling is impossible at the application layer. One way to get around it is to run a separate connection pool, like PgBouncer, as an intermediary between your functions and your database.

These workarounds can be heavy-handed, and it's why you see the popularity of tools like DynamoDB in serverless architectures. DynamoDB has an HTTP-based connection model, rather than a persistent TCP connection like relational databases.

DynamoDB is an excellent tool, and I'm a huge fan (check out my prior content on DynamoDB modeling patterns or the DynamoDB Guide). That said, it's not the right tool for every situation.

If you're in the early stages of a project, you're in a bit of a pickle. Serverless compute is likely the answer due to the speed of development and low maintenance burden but locking into an IE system is premature. You usually want a more flexible datastore at that point.

There's still not a great option for serverless users that want a traditional PE system. I remain hopeful that Serverless Aurora is the answer, but it's clear we're not there yet.

Conclusion

The PIE theorem is a handy way to think about your data needs and choose the right tool for the job. In this post, we discussed the PIE theorem and its three main permutations. We also learned why the serverless paradigm is still an awkward fit in the database world. Even so, I'm hopeful about the future of databases as they adjust to a world of hyper-ephemeral compute.

Published 22 Jan 2019

Database

AWS Data Hero providing training and consulting with expertise in DynamoDB, serverless applications, and cloud-native technology.
Alex DeBrie on Twitter

ALSO ON **ALEXDEBRIE**

Three ways to use AWS services from a …
a year ago • 5 comments
By default, Lambda functions in a VPC cannot access the public …

Three Projects to Get You Started with …
2 years ago
Serverless is one of the hottest programming trends in 2019. Check out three …

SQL, NoSQL, and Scale: How …
a year ago • 14 comments
Understand when and why relational databases don't scale, and how …

A I
AW
2 ye
Loc
AW
unc

What do you think?

10 Responses

Upvote Funny Love Surprised Angry Sad

1 Comment **alexdebrie** **Disqus' Privacy Policy** 1 **Login**

Recommend Tweet Share **Sort by Best**

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Yeswanth kumar Bandaru • 2 months ago
Very nice article and easy to follow. just one small doubt can you please throw some light on how Normalization leads to pattern flexibility in Relational Databases.
• Reply • Share ›

Subscribe **Add Disqus to your siteAdd DisqusAdd** **Do Not Sell My Data**