Use the best data store for the job

08/30/2018 • 2 minutes to read • Contributors

In this article

Pick the storage technology that is the best fit for your data and how it will be used

Gone are the days when you would just stick all of your data into a big relational SQL database. Relational databases are very good at what they do — providing ACID guarantees for transactions over relational data. But they come with some costs:

- Queries may require expensive joins.
- Data must be normalized and conform to a predefined schema (schema on write).
- Lock contention may impact performance.

In any large solution, it's likely that a single data store technology won't fill all your needs. Alternatives to relational databases include key/value stores, document databases, search engine databases, time series databases, column family databases, and graph databases. Each has pros and cons, and different types of data fit more naturally into one or another.

For example, you might store a product catalog in a document database, such as Cosmos DB, which allows for a flexible schema. In that case, each product description is a self-contained document. For queries over the entire catalog, you might index the catalog and store the index in Azure Search. Product inventory might go into a SQL database, because that data requires ACID guarantees.

Remember that data includes more than just the persisted application data. It also includes application logs, events, messages, and caches.

Recommendations

Don't use a relational database for everything. Consider other data stores when appropriate. See Choose the right data store.

Embrace polyglot persistence. In any large solution, it's likely that a single data store technology won't fill all your needs.

Consider the type of data. For example, put transactional data into SQL, put JSON documents into a document database, put telemetry data into a time series data base, put application logs in Elasticsearch, and put blobs in Azure Blob Storage.

Prefer availability over (strong) consistency. The CAP theorem implies that a distributed system must make trade-offs between availability and consistency. (Network partitions, the other leg of the CAP theorem, can never be completely avoided.) Often, you can achieve higher availability by adopting an *eventual consistency* model.

Consider the skillset of the development team. There are advantages to using polyglot persistence, but it's possible to go overboard. Adopting a new data storage technology requires a new set of skills. The development team must

understand how to get the most out of the technology. They must understand appropriate usage patterns, how to optimize queries, tune for performance, and so on. Factor this in when considering storage technologies.

Use compensating transactions. A side effect of polyglot persistence is that single transaction might write data to multiple stores. If something fails, use compensating transactions to undo any steps that already completed.

Look at bounded contexts. *Bounded context* is a term from domain driven design. A bounded context is an explicit boundary around a domain model, and defines which parts of the domain the model applies to. Ideally, a bounded context maps to a subdomain of the business domain. The bounded contexts in your system are a natural place to consider polyglot persistence. For example, "products" may appear in both the Product Catalog subdomain and the Product Inventory subdomain, but it's very likely that these two subdomains have different requirements for storing, updating, and querying products.