

New

Announcing MongoDB Atlas Vector Search and Dedicated Search Nodes for genAI use cases

Building with Patterns: A Summary

[Learn More About MongoDB at MongoDB University](#)**Daniel Coupal** and **Ken W. Alger**

April 27, 2019 | Updated: October 3, 2023

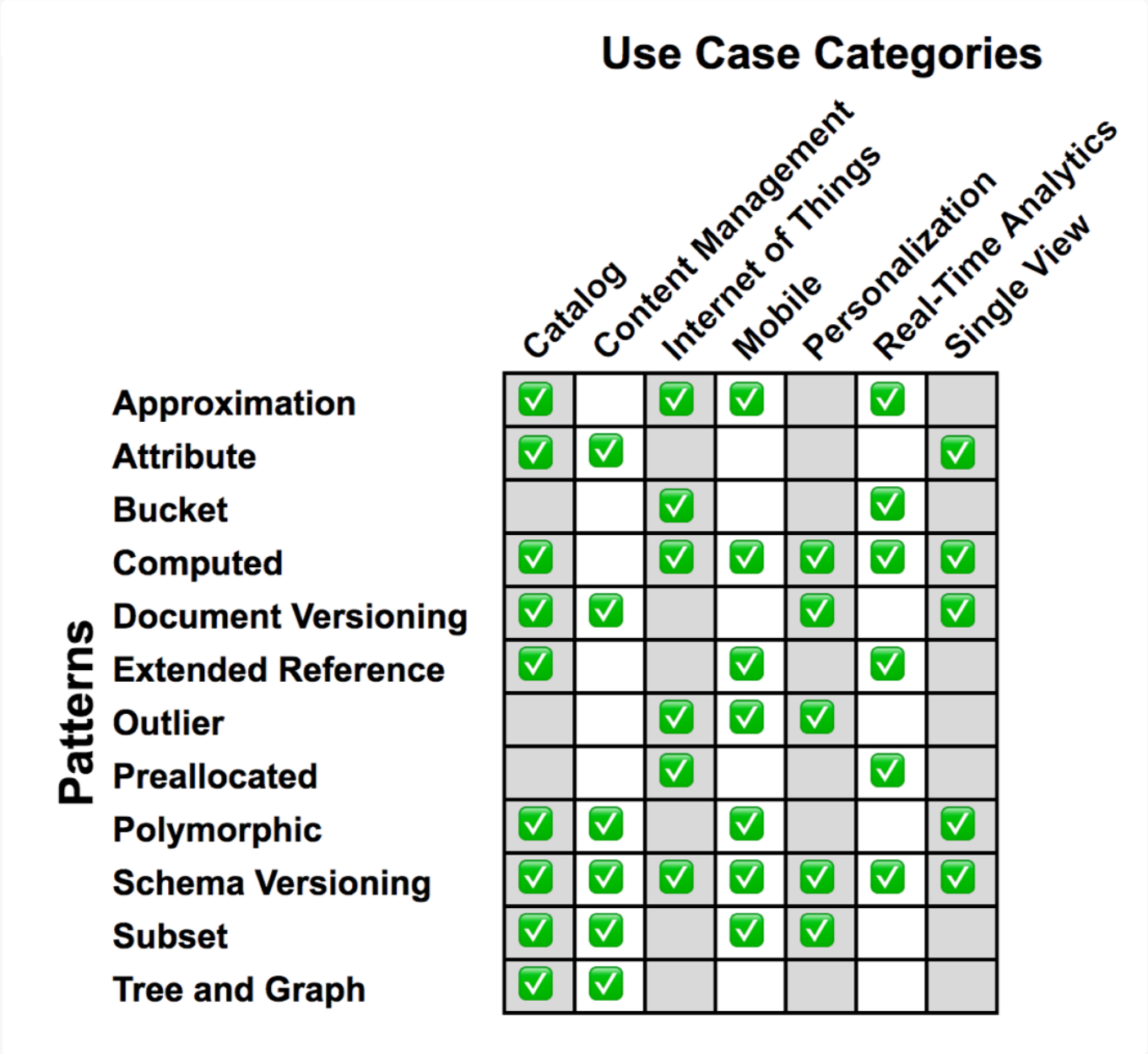
#Developer #University

This post is also available in: [Deutsch](#), [Français](#), [Español](#), [Português](#)

As we wrap up the *Building with Patterns* series, it's a good opportunity to recap the problems the patterns that have been covered solve and highlight some of the benefits and trade-offs each pattern has. The most frequent question that is asked about schema design patterns, is "I'm designing an application to do X, how do I model the data?" As we hope you have discovered over the course of this blog series, there are a lot of things to take into consideration to answer that. However, we've included a *Sample Use Case* chart that we've found helpful to at least provide some initial guidance on data modeling patterns for generic use cases.

Sample Use Cases

The chart below is a guideline for what you have found after years of experience working with our customers of what design patterns are used in a variety of applications. This is not a “set in stone” set of rules about which design pattern can be used for a particular type of application. Ensure you look at the ones that are frequently used in your use case. However, don't discard the other ones, they may still apply. How you design your application’s data schema is very dependent on your data access patterns.



Design Pattern Summaries

Approximation

The **Approximation Pattern** is useful when expensive calculations are frequently done and when the precision of those calculations is not the

highest priority.



Pros

- Fewer writes to the database.
- Maintain statistically valid numbers.

Cons

- Exact numbers aren't being represented.
- Implementation must be done in the application.

Attribute

The **Attribute Pattern** is useful for problems that are based around having big documents with many similar fields but there is a subset of fields that share common characteristics and we want to sort or query on that subset of fields. When the fields we need to sort on are only found in a small subset of documents. Or when both of those conditions are met within the documents.

Pros

- Fewer indexes are needed.
- Queries become simpler to write and are generally faster.

Bucket

The **Bucket Pattern** is a great solution for when needing to manage streaming data, such as time-series, real-time analytics, or Internet of Things (IoT) applications.

Pros

- Reduces the overall number of documents in a collection.
- Improves index performance.
- Can simplify data access by leveraging pre-aggregation.

Computed

When there are very read intensive data access patterns and that data needs to be repeatedly computed by the application, the **Computed Pattern** is a great option to explore.

Pros

- Reduction in CPU workload for frequent computations.
- Queries become simpler to write and are generally faster.

Cons

- It may be difficult to identify the need for this pattern.
- Applying or overusing the pattern should be avoided unless needed.

Document Versioning

When you are faced with the need to maintain previous versions of documents in MongoDB, the **Document Versioning** pattern is a possible solution.

Pros

- Easy to implement, even on existing systems.
- No performance impact on queries on the latest revision.

Cons

- Doubles the number of writes.
- Queries need to target the correct collection.

Extended Reference

You will find the **Extended Reference** pattern most useful when your application is experiencing lots of JOIN operations to bring together frequently accessed data.

Pros

- Improves performance when there are a lot of JOIN operations.
- Faster reads and a reduction in the overall number of JOINS.

Cons

- Data duplication.

Outlier

Do you find that there are a few queries or documents that don't fit into the rest of your typical data patterns? Are these exceptions driving your

application solution? If so, the **Outlier Pattern** is a wonderful solution to this situation.



Pros

- Prevents a few documents or queries from determining an application's solution.
- Queries are tailored for "typical" use cases, but outliers are still addressed.

Cons

- Often tailored for specific queries, therefore ad hoc queries may not perform well.
- Much of this pattern is done with application code.

Pre-allocation

When you know your document structure and your application simply needs to fill it with data, the **Pre-Allocation Pattern** is the right choice.

Pros

- Design simplification when the document structure is known in advance.

Cons

- Simplicity versus performance.

Polymorphic

The **Polymorphic Pattern** is the solution when there are a variety of documents that have more similarities than differences and the documents need to be kept in a single collection.

Pros

- Easy to implement.
- Queries can run across a single collection.

Schema Versioning

Just about every application can benefit from the **Schema Versioning Pattern** as changes to the data schema frequently occur in an application's lifetime. This pattern allows for previous and current versions of documents to exist side by side in a collection.

Pros

- No downtime needed.
- Control of schema migration.
- Reduced future technical debt.



Cons

- Might need two indexes for the same field during migration.

Subset

The **Subset Pattern** solves the problem of having the working set exceed the capacity of RAM due to large documents that have much of the data in the document not being used by the application.

Pros

- Reduction in the overall size of the **working set**.
- Shorter disk access time for the most frequently used data.

Cons

- We must manage the subset.
- Pulling in additional data requires additional trips to the database.

Tree

When data is of a hierarchical structure and is frequently queried, the **Tree Pattern** is the design pattern to implement.

Pros

- Increased performance by avoiding multiple JOIN operations.

Cons

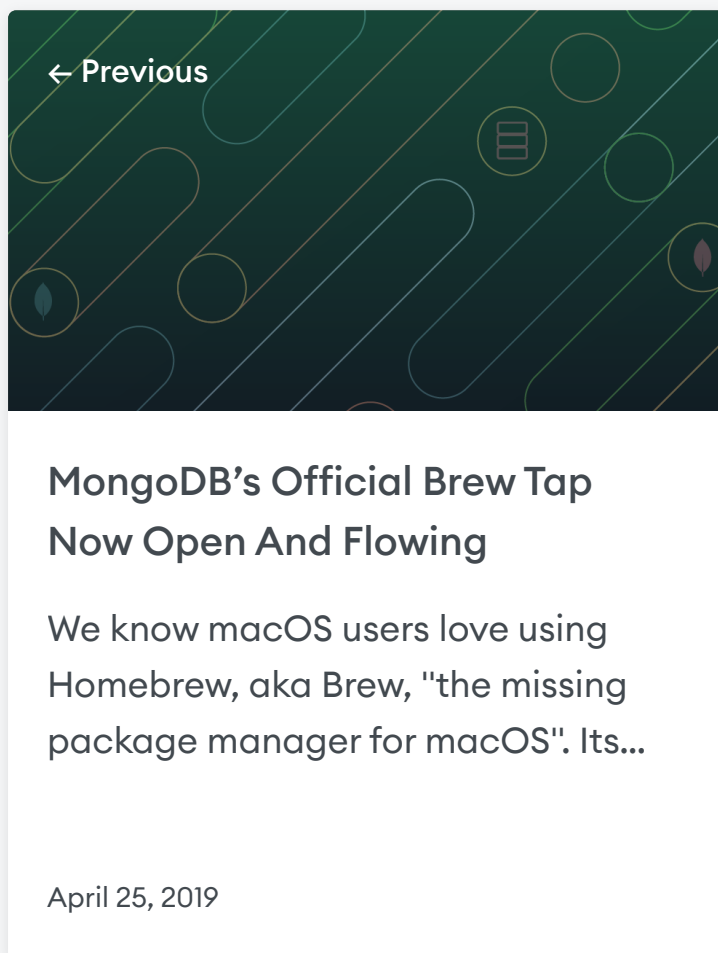
- Updates to the graph need to be managed in the application.

Conclusion

As we hope you have seen in this series, the MongoDB document model provides a lot of flexibility in how you model data. That flexibility is incredibly powerful but that power needs to be harnessed in terms of your application's data access patterns. Remember that schema design in MongoDB has a

tremendous impact on the performance of your application. We've found that performance issues can frequently  to poor schema design.

Keep in mind that to further enhance the power of the document model, these schema design patterns can be used together, when and if it makes sense. For example, Schema Versioning can be used in conjunction with any of the other patterns as your application evolves. With the twelve schema design patterns that have been covered, you have the tools and knowledge needed to harness the power of the document model's flexibility.



[Next →](#)

Leveraging MongoDB Atlas in your Internal Developer Platform (IDP)

DevOps, a portmanteau of “Developer” and “Operations”, rose to prominence around the early 201...

January 4, 2024



About

[Careers](#)[Investor Relations](#)[Legal Notices](#)[Privacy Notices](#)[Security Information](#)[Trust Center](#)

Support


[Contact Us](#)[Customer Portal](#)


Atlas Status


Customer Support


X


Social


 GitHub


 Stack Overflow

 LinkedIn

 YouTube

 Twitter

 Twitch

 Facebook

© 2023 MongoDB, Inc.