RED PILL or BLUE PILL

DOCUMENT DATABASE MODELING FOR RELATIONAL MODELERS

https://www.theguardian.com/film/2019/jan/21/from-red-pills-to-red-white-and-blue-brexit-how-the-matrix-shaped-our-reality

SPEAKER BIO

- BILL COULAM
- EST 1995
  - CONSULTANT AND EMPLOYEE IN SAN FRANCISCO, DENVER, HOUSTON, VIRGINIA AND UTAH
  - TELECOM, ENERGY, FITNESS, PROPERTY INSURANCE, HEALTHCARE, FOR-PROFIT, NON-PROFIT
  - SOFTWARE ENGINEER TO DATA ENGINEER, DATA MODELER
  - DATA AND DATABASE ARCHITECT FOR OVER 25 YEARS (SENIOR, PRINCIPAL & ENTERPRISE)
- SPEAKER AT ORACLE CONFERENCES AND LOCAL SEMINARS SINCE 2001
- CURRENTLY SENIOR DATA ARCHITECT AT FMG GLOBAL IN RHODE ISLAND (LARGEST COMMERCIAL PROPERTY INSURER ON THE PLANET)

CONSULT, MODEL, TRAIN, ANALYZE, TUNE

dbsherpa.com

AGENDA

RELATIONAL DESIGN [5]

DOCUMENT DESIGN

- WARNINGS [5]
- CHECKLIST [5]
- MODELING GUIDE [10]
- ENGINEERING GUIDE [5]
  - MONGODB TIPS [5]

Q&A [10]

RELATIONAL DESIGN

The Design Process

1. Research entities
2. Research attributes
3. Research relationships
4. Research cardinality and optionality
5. Research business & data rules
6. Create conceptual/logical model
7. Normalize and apply naming standard
8. Create physical model and apply naming standard. Any denormalizations + temporal?
9. Check the model into the project repository
10. Have the model reviewed and address issues
11. Implement and test the model in the target database
12. Generate DDL and check in project repository
13. Publish the model diagrams and data dictionary

Send email to bill.coulam@dbsherpa.com for full article

RELATIONAL DESIGN

Guiding Principles

- Know Your Data
- Model First, Cleanly and Thoroughly
- Duplication is a Disease
- Name Things Well
- Keep it Simple
- Protect Your Data
- Unleash Your Database
- Design to End Goals from Day 1

Send email to bill.coulam@dbsherpa.com for full article

RELATIONAL DESIGN

TAKING THE TIME TO REALLY UNDERSTAND THE DATA IS THE MOST IMPORTANT ASPECT OF DESIGNING AN ENTERPRISE-GRADE, RESILIENT, ACCURATE, SECURE DATABASE

Example Questions for Entities & Sets:

RELATIONAL DESIGN

Business Analysis for Data Modeling

- Will that report be paginated, or will you want the entire query results at once?
- How many records do you initially anticipate in the table behind that screen?
- How much will that grow per day/month/year?
- Which fields identify a unique instance of this record?
- How long does the customer/business anticipate the data should stay online and accessible? Once the retention period is past, could the data be deleted, or must it be soft-deleted (inactivated), put in historical tables, or moved to a near-line storage medium?
- Could this app and its end users tolerate eventual consistency, or do they demand immediate accuracy?
- What questions and queries are anticipated on this data the app will store?
- Is the creation and last modification user and timestamp sufficient, or do we need to record metadata and before/after values for every change?
- What response times are acceptable for this event/action/report/page request?
- Have you identified enough detail about the entity to satisfy requirements? (think of a used car app where only the VIN, miles, condition, make, model and year need to be known to post, but there might be far more optional and required fields to make the app usable)

Example Questions for Fields:

RELATIONAL DESIGN

Business Analysis for Data Modeling

- What is the maximum amount of characters allowed in that field?
- Is that field optional or required?
  - Is it required only upon certain condition(s)?
  - Should it be empty upon certain condition(s)?
- Should the data be normalized to uppercase on the backend (think addresses), or stored exactly as the user typed it in?
- What are the allowable values for that radio/checkbox/listbox field?
- Is there a required format, or set of allowable formats, for the values in the field that should be checked by the client, the database, or both?
- Will that textbox text ever be searched or mined, or will it just be displayed for other users to read?
- How many digits of precision are required for that number?
- Is date sufficient, or should time be recorded as well?
- Will that field/label need to support other languages?
- Does that field need to be masked during entry, or encrypted at rest or in transit across the network?

RELATIONAL DESIGN

Business Analysis for Data Modeling

Example Questions for Relationships:

- What are the business rules surrounding how many children or parents are allowed?
- Is the relationship optional or required? Is the required…ness dependent on certain conditions?
- Is the relationship transferrable? (can it move from one owner to another)
- Where the relationship is recursive (for example employees managed by senior employees), are there business rules about when the field containing the FK value can be filled, or when it should be empty? (for example an employee may not be supervisor of themselves)
- …and so forth

RELATIONAL DESIGN

Sample Model: Identity

Homework: A simple library checkout application

Highland City Library

highlandcitylibrary.org
801-772-4528

- Checkout Receipt -

Patron Barcode: **********901

Number of items:

Barcode: 35400000660630
Title: War horse
Due: 03/04/2023

Barcode: 35400000660523
Title: Regarding the fountain : a tale, in letters, of liars and leaks
Due: 03/04/2023

Barcode: 35400000444167
Title: Snow treasure
Due: 03/04/2023

02/11/2023 01:34:46 PM

You have been asked to model the data structures for a library check-out system.

Have a look at the following checkout receipt and write down the entities, attributes and relationships that you foresee the system needing.

Then write down any questions that come to mind as you are teasing those out of this receipt and asking yourself if you understand everything about the needs of this system.

DO NOT PROCEED TO THE NEXT SLIDE.
ACTUALLY DO THIS EXERCISE.

Bonus: Create a conceptual (subject area) model, a logical model, and a physical model before the next slide

Homework: A simple library checkout application

Highland City Library
highlandcitylibrary.org
801-772-4528

- Checkout Receipt -

Patron Barcode: **********901

Number of items:

Barcode: 35400000660630
Title: War horse
Due: 03/04/2023

Barcode: 35400000660523
Title: Regarding the fountain : a tale, in letters, of liars and leaks
Due: 03/04/2023

Barcode: 35400000444167
Title: Snow treasure
Due: 03/04/2023

02/11/2023 01:34:46 PM

Entities & Attributes:

- Receipt (Library, Patron Barcode, [Item], Timestamp in local TZ)
- Library (Name, URL, Phone)
- Patron (Barcode)
- Item (Barcode, Type, Title, Due Date)

Relationships:

- Library issues a card to a patron. When checking out items, patron receives a receipt. Receipt lists 1..25 items

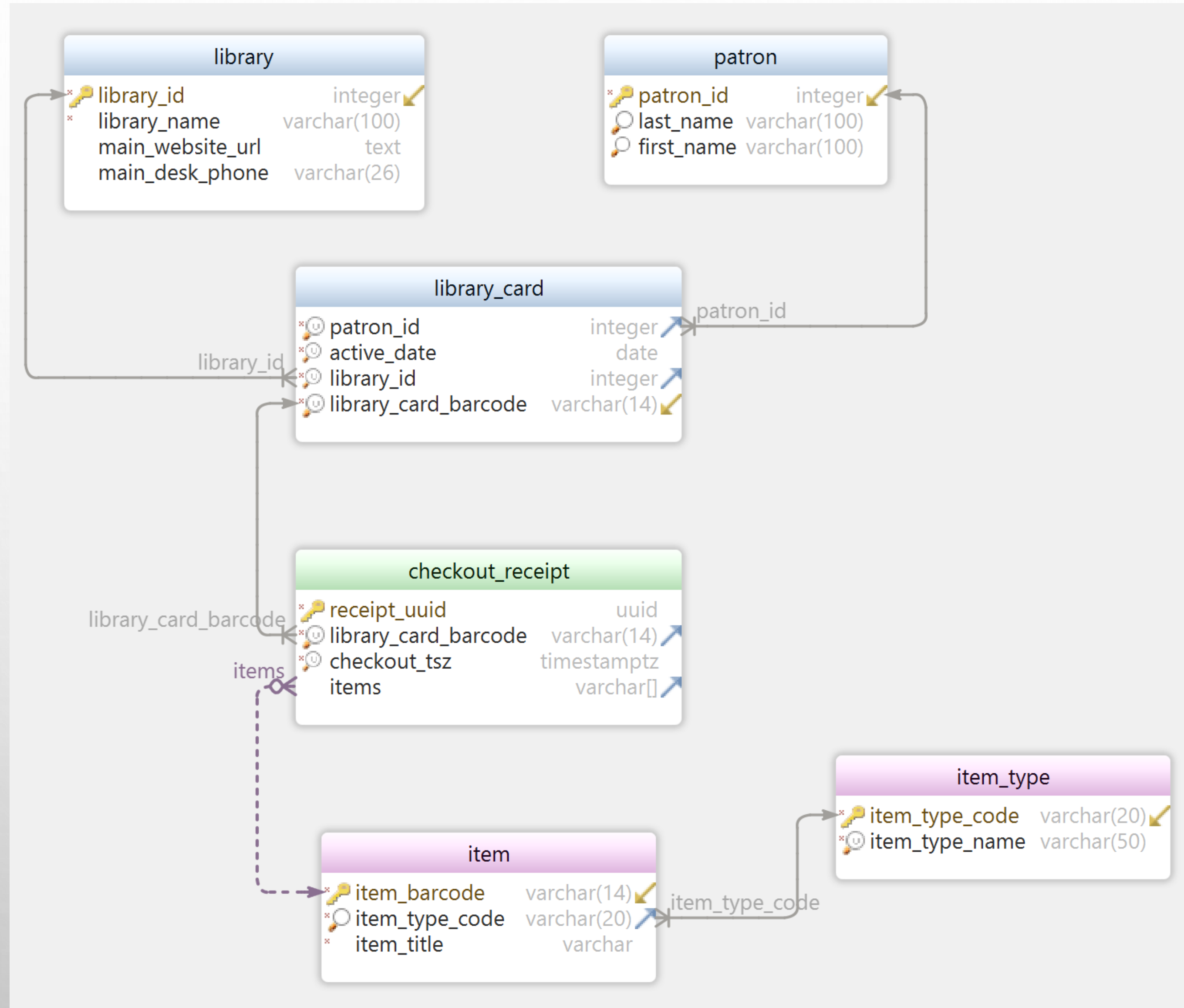
Rules:

- Books and Audiobooks are due in 3 weeks
- Videos are due in 1 week
- Patron may checkout a maximum of 25 total items

Questions (small sample):

- Would you like to store and display the item type?
- How old must a patron be to own a card?
- Can the card be used at other city libraries?
- Is Barcode always numeric? Can it start with 0's? Does it have a fixed length?

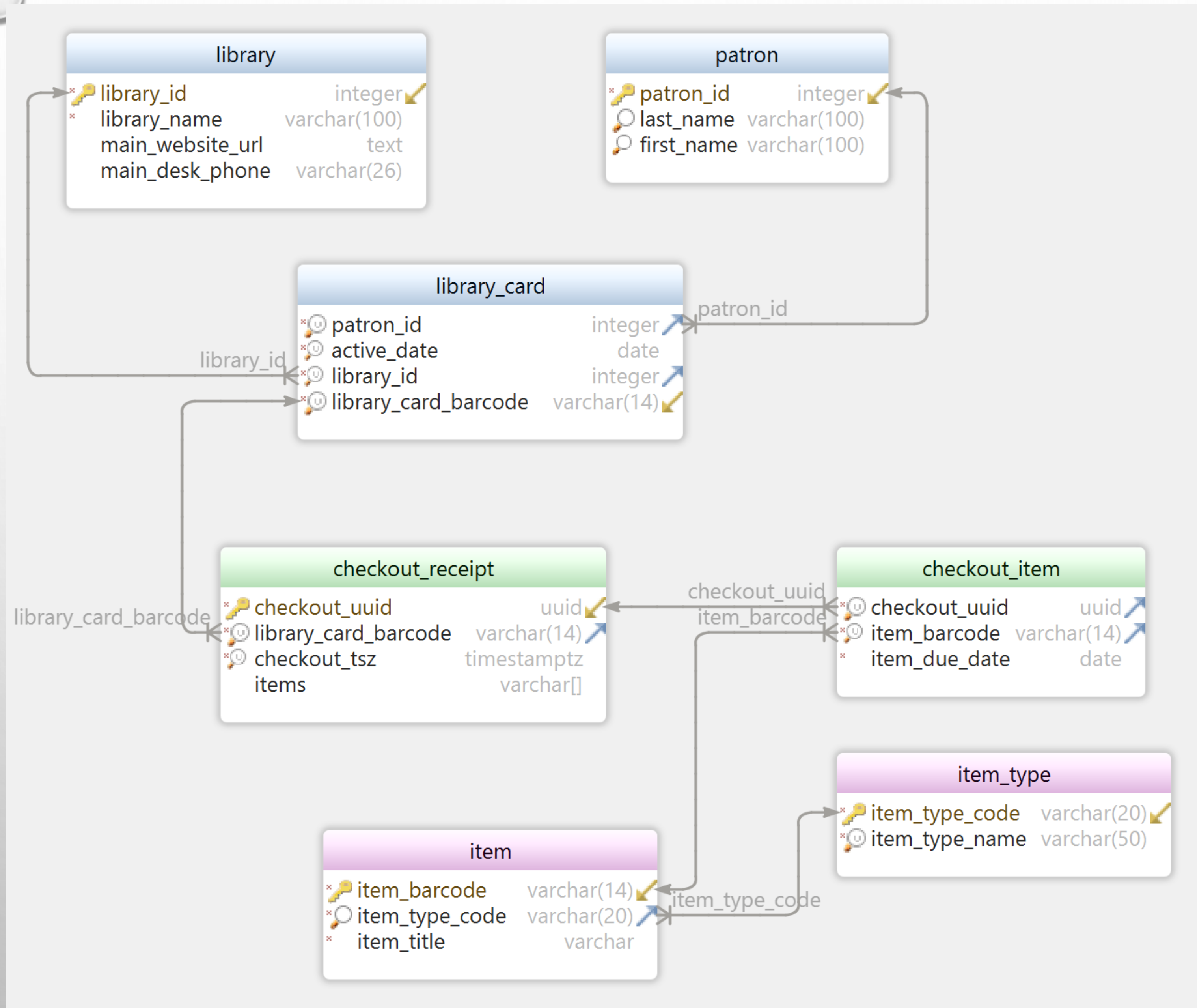
Homework: A simple library checkout application



What is missing?

Do you anticipate problems with this model as you walk a patron through getting a library card and checking out a mix of items?

Homework: A simple library checkout application



What about this model?

Due dates are now handled cleanly.

Searching for items in currently checked out inventory is easy.

What holes can you still see?

RELATIONAL DESIGN

Physical Model Standards Checklist

- ❑Each table, view, materialized view and column has a clear and useful explanation filled by a comment statement
- ❑Each table has a single column surrogate (primary) key constraint
- ❑Each table has a unique (natural) key constraint, unless duplicates are expected and welcomed
- ❑Each table conforms to third normal form
- ❑Each table and column name should conform to the Naming Guide
- ❑Valid value lists are kept in parent reference/lookup tables, not enums or check constraints
- ❑Tables include the four standard audit columns
- ❑Tables may not include personally identifiable or sensitive data
- ❑Downstream teams (typical DW, BI and Analytics) or data contracts have been notified of any changes being made to data, interfaces, APIs, or agreements
- ❑Parent primary key values found in child tables are protected by an explicit foreign key on the child column(s). Each foreign key constraint is supported by an index. Each foreign key column name matches its parent key column name.

Send email to bill.coulam@dbsherpa.com for full article

RELATIONAL DESIGN

Physical Model Naming Recommendations

- ❑Tables: Each entity (table) name is singular, not plural
- ❑Tables: Use a common prefix for related tables to create a logical grouping
- ❑Tables: Use a type suffix to designate table type*, when applicable, like AUD, HIST, TYPE, STG, etc.
- ❑DB Objects: Use a suffix to designate the object type* (i.e. FK, PK, IX, SQ, VW, MV)
- ❑Indexes: Utilize the index type suffix* to identify index attributes and implementation (i.e. unique, function-based, b-tree, brin, gin, etc.)
- ❑Constraints: If not using default constraint names, ensure the name is readable
- ❑Triggers: Utilize the type suffix* to indicate the firing events and timing

* See the Naming Guide for recommended type codes, format and abbreviations

RELATIONAL DESIGN

Physical Modeling Recommendations

- ❑ Manually populated surrogate keys should avoid using numbering ranges or schemes
- ❑ Surrogate keys should have no intrinsic business meaning
- ❑ Tables should avoid columns of repeated attributes (email1, email2, email3) or array columns that hold multiple values
  - ❑ Separate repeating attributes into a child table with a type indicator to distinguish between them
- ❑ All SQL/DDL/migration scripts should be readable and well-formatted

  (Most IDEs will have an option to auto-format DDL and DML, making this easy to achieve)

Send email to bill.coulam@dbsherpa.com for full article

RELATIONAL DESIGN

Physical Modeling Considerations

- How much data will there be? Will size and concurrency demand a partitioning scheme to meet performance goals?
- How long will it be kept?
- Will old data be soft-deleted, shuffled to historical tables, archived or simply removed?
- Will privileged users or reports want, or need to access soft-deleted, shuffled or archived data? How often?
- How will deletions/truncations take place when pruning?

SIDE-BY-SIDE COMPARISON

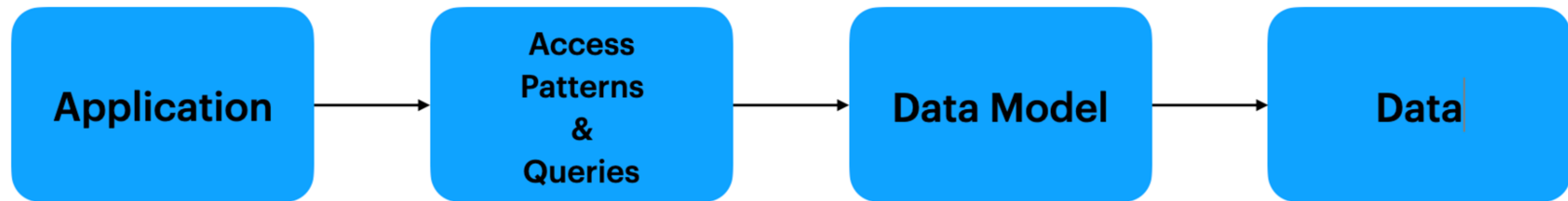
Design Approach

Start with

Relational Modeling

Start with

NoSQL Modeling



DOCUMENT DESIGN

"It is true that lack of schema increases agility for the engineer who owns putting data into the system. However, it kicks the problem down to the readers of the data, who are usually an order of magnitude greater in number and often don't have the context about the state of the data when it was written. These users are usually the ones who are generating value from that data and should have as few roadblocks as possible.

To give an analogy, imagine libraries saying they are doing away with the Dewey Decimal System and just throwing the books into a big hole in the ground and declaring it a better system because it is way less work for the librarians. There is a time and place for semi-structured data, because sometimes you don't know the shape of some of the data ahead of time or it is sparsely populated. But if you truly don't understand any of the data coming [in] or what it will look like, then what good is it?

The truth is that there is always schema. The data always makes sense to someone. That someone should take the time to encode that knowledge into the platform so it is usable by the next people. If it is a mix of data that is understood and some that is changing rapidly, put the latter into a semi-structured column in a database, then figure out what columns to project out of it later."

- Rick Negrin, VP of Product Management, Singlestore

DOCUMENT DESIGN

Warnings from the “Trenches”

"In many ways, up-front data design with NoSQL databases can actually be *more* important than it is with traditional relational databases... Beyond the performance topic, NoSQL databases with flexible schema capabilities similarly require *more* discipline in aligning to a common information model... The flexible schema is a great innovation for quick evolution of your data model, and yet it requires discipline to harvest the benefits without experiencing major data quality issues and frustrations as a result.”

- Ryan Smith, Information Architect at Nike (now Sr. Mgr Data Lake Product and Program Management with AWS)

DOCUMENT DESIGN

Warnings from the "Trenches"

- Flexibility is its greatest feature and its greatest drawback
- Best used as a denormalized cache of structured/relational content for lightning fast reads
- ACID transactions and bulk updates are supported, but slow
- Devs have to manage relationships and data integrity, instead of DBAs
- No support for SQL language or tools. Reporting out of document DBs is difficult.
- Do not try to make a document DB look relational (normalized). If the data is structured (80 to 90% of business data is), start with relational. Use a JSON column for the unstructured bits.
- Data modeling and data management for document DBs demands MORE rigor from devs, not less

DOCUMENT DESIGN

Warnings from the “Trenches”

- Intentionally plan, research and consider the performance and quality implications of object composition/projection (embed) vs. object references
- Aggregation pipelines and searching across collections is difficult to grok and implement, even for seasoned document DB devs.
- All the duplication and denormalization must be meticulously kept in sync (it usually is not)
- You will probably need another layer ON TOP of your document DB to extract business intelligence value out of the document data. Budget and plan for this additional layer (server, software, tools, people and maintenance)
- Use replace, not update (forces index re-write)

DOCUMENT DESIGN

When is Document the Right Choice?

- Personal projects, rapid prototyping and POC's where little is known about the eventual application and requirements change frequently
- Data is unstructured or semi-structured
- Content and context are dynamic (different shape per record)
- If every nano-second counts and IOs must be kept to a bare minimum
- The following do not count as good reasons: ~~Because I used it at school~~. ~~Because it just works with JSON~~. ~~Because my last company used it~~. ~~Because I don't know SQL~~. ~~Because joins are slow and awful~~.

Send email to bill.coulam@dbsherpa.com for full article

DOCUMENT DESIGN

Common Uses Cases for Document

Personalization Subsystems	Data that changes very little or not at all after it is written (WORMy data)	Aggregated Enterprise Caches
• User Preferences • Site Preferences (white-label framework for custom content, themes, color, logo, style, etc.)	• e.g. wide/denormalized IoT records that are never or rarely updated • activity, system, and error logs • time-series • payment processing	• Single view (like Customer360) • Enterprise Data Hub

DOCUMENT DESIGN

- OK, so you are aware of all the pitfalls and dangers and you are sure that a document solution is still the right choice.
- What now? How do I design and build to a document information model in all the right ways?
- We'll look at
  - JSON document column (briefly)
  - JSON document database

JSON COLUMN

- Keep the structured data as regular columns
- Put the flexible stuff, the fields that could change shape per record, in the JSON column

Oracle (21c+)	Postgres (9.4+)
create table purchase_order (po_id integer not null, po_ts timestamp with time zone not null, po_doc **json**, constraint purchase_order_pk (po_id) primary key);	create table if not exists purchase_order (po_id bigint not null, po_ts timestamptz not null, po_doc **jsonb**, constraint purchase_order_pk(po_id) primary key);

AskTom JSON in the Database "Office Hours" site

JSON DOCUMENT DATABASE

(MOST OF WHAT FOLLOWS IS HEAVILY SKEWED TO MONGODB AND MONGODB ATLAS)

Rank			DBMS	Database Model	Score		
Feb 2023	Jan 2023	Feb 2022			Feb 2023	Jan 2023	Feb 2022
1.	1.	1.	MongoDB	Document, Multi-model	452.77	-2.42	-35.88
2.	2.	2.	Amazon DynamoDB	Multi-model	79.69	-1.87	-0.67
3.	3.		Databricks	Multi-model	60.33	-0.49	
4.	4.	↓ 3.	Microsoft Azure Cosmos DB	Multi-model	36.51	-1.45	-3.45
5.	5.	↓ 4.	Couchbase	Document, Multi-model	24.86	-0.44	-5.21
6.	6.	↓ 5.	Firebase Realtime Database	Document	18.49	-0.27	-0.66
7.	7.	↓ 6.	CouchDB	Document, Multi-model	14.45	-0.44	-3.01
8.	8.	↑ 9.	Google Cloud Firestore	Document	11.51	+0.41	+2.45
9.	9.	↓ 7.	MarkLogic	Multi-model	8.84	-0.01	-0.61
10.	10.	↓ 8.	Realm	Document	8.24	-0.09	-1.17

https://db-engines.com/en/ranking/document+store

SIDE-BY-SIDE COMPARISON

Data Objects and Elements

Relational Database/Data	Document Database/Data
Database	Database
Table	Collection
Index	Index
Row	Document
View	View
Column	Field
Join	Embed or Link

DOCUMENT DESIGN

Guiding Principles

- Know Your Data & Your Application
  - ***Must*** *know data needs and access patterns*
- Model First, Cleanly and Thoroughly
- Duplication is Welcome
  - *But design real-world objects that naturally contain and embed other data*
- Name Things Well
  - *Stick with snake_case*
- Keep it Simple
- Protect Your Data
  - *Use JSON Schema for validation*
- Unleash Your Database
  - *But balance read and write performance with consistency*
- Design to End Goals from Day 1

Send email to bill.coulam@dbsherpa.com for full article

DOCUMENT DB MODELING GUIDE

Schema Design Rules of Thumb

One: Favor embedding unless there is a compelling reason not to

Two: Needing to access an object on its own is a compelling reason not to embed it

Three: Arrays should not grow without bound. If there are more than a couple of hundred documents on the "many" side, don't embed them; if there are more than a few thousand documents on the "many" side, don't use an array of ObjectID references, instead link to the parent doc ID in the "many" side. High-cardinality arrays are a compelling reason not to embed.

Four: Don't be afraid of application-level joins: if you index correctly and use the projection specifier then application-level joins are barely more expensive than server-side joins in a relational database.

Five: Consider the write/read ratio when denormalizing. A field that will mostly be read and only seldom updated is a good candidate for denormalization: if you denormalize a field that is updated frequently then the extra work of finding and updating all the instances is likely to overwhelm the savings that you get from denormalizing.

Six: As always with MongoDB, how you model your data depends – entirely – on your particular application's data access patterns. You want to structure your data to match the ways that your application queries and updates it.

Adopted from https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design
and https://www.youtube.com/watch?v=leNCfU5SYR8

DOCUMENT DESIGN

Physical Model Standards Checklist

Analysis

- ❑ Performed full analysis of the application/service problem domain, business context, and data elements. What is the data, and how will the application use it?
  - ❑ List of foreseeable data queries and operations (and preferably the interface or API surrounding each)
  - ❑ List of data requirements: consistency, response time expected of each query and operation, quality, security & privacy, retention, expected volume and CRUD velocity, expected growth, expected peak usage and duration

Database

- ❑ Each database is named using lowercase words separated by hyphens, typically ending in -db or -service or -store

Send email to bill.coulam@dbsherpa.com for full article

DOCUMENT DESIGN

Physical Model Standards Checklist

Collections

- ❑ Each collection is named using snake_case
- ❑ Draft and publish a Collection-Relationship Diagram within your modeling tool
- ❑ Each collection should have a verbose and useful description persisted in the modeling tool*
- ❑ Create a unique index for each collection's natural key (an entity that permits and welcomes duplicates is very rare)
- ❑ Each collection should have a field to track schema versioning. We recommend "schema_version" for this field. It is clear and concise.
- ❑ Each choice to embed subdocuments, or link them using a document ID reference, has been deliberately and carefully pondered
- ❑ Is there any denormalization that needs to happen to support read requirements?
  - ❑ Document the plan and code to keep the denormalized fields updated if the master changes
- ❑ The common document design patterns have been considered and used where applicable.

Send email to bill.coulam@dbsherpa.com for full article

DOCUMENT DESIGN

Physical Model Standards Checklist

Fields

- Each field follows the Naming Guide,
- Generic field names are not allowed (except for the automatic "_id" field)
  - Keep it short, but readable and intuitive
  - Use standard domain abbreviations when applicable (_code, _id, _date, _num, _name, etc.)
- Each field has been researched until understood, and the following attribute prompts can be filled in:
  - Data type
  - Length (open vs restricted for business rules or regulations; min length, max length)
  - Optionality (is it always required, or can it ever be empty?)
  - Cardinality (how many of X are likely, what is the maximum of X possible?)
  - Sensitivity (personally, legally or financially? which roles are allowed to vew/modify?)
  - Default value?
  - Valid values (including source of master valid values list)
    - Numeric: how many digits of precision, if any?
    - Date/Time: Just the date, or should full timestamp be kept?
  - Used as a filter or sort element? (see Secondary Indexes section)
  - Normalization and formatting rules? (like addresses, phones, SSNs and emails)

Send email to bill.coulam@dbsherpa.com for full article

DOCUMENT DESIGN

Physical Model Standards Checklist

Data Integrity

- ❑ Ensure data integrity and consistency are protected. Use JSON Schema validation to validate the rules and formatting in the Fields section above

Secondary Indexes

- ❑ Ensure each collection has one or more indexes which support the expected queries upon that collection.

Views

- ❑ Complex queries and aggregation pipelines have been made re-usable by being placed behind a view*.

Scaling Options

- ❑ Have considered replication vs sharding* vs vertical expansion as scaling solutions for the service

DOCUMENT DB MODELING GUIDE

The Design Process

- **Analyze and document** the application prior to modeling
- **Design the model**: distill and model the documents and their relationships
  - Separate logical groups and distinct data requests into collections.
  - Build a draft of each collection as a modeled document, crafting each field, reference, array and embedded subdocument
  - Apply common modeling patterns, as applicable
  - Build and publish a CRD (Collection-Relationship Diagram) for comment and correction
- **Protect the model**: build a JSON Schema validator for each document
- **Implement the model**
  - Design, configure and create the database
  - Create the collections, views and indexes
  - Test and measure with sunny data, dirty data and production loads.
- **Version the model**: iterate on the documents and indexes until application requirements are met

See https://www.youtube.com/watch?v=DUCvYbcgGsQ

DOCUMENT DB MODELING GUIDE

Analyze and Document

- Research your data AND your application's data needs
- Can begin coding without a perfect understanding. Evolve the schema as more is learned.
- Interview those who know the most about what the application should do. Obtain:
  - current and future functionality to determine the CRUD operations required, what data will be involved in each, and the relative significance of each
  - is the field optional or required. If required, is there a default value, list of valid values and a valid format?
  - response time, quality and security expectations/contracts
  - data retention policies and parameters
  - data volume - how much of each document and field is expected and growth projections
  - data velocity - how fast will the data arrive, and expected concurrency
  - data variety - structured, unstructured, mix of both? binary fields? huge text fields? datatypes?
  - data workloads - usage patterns, expected growth trends, peak usage times and duration, size and shape of user base

ARMED WITH THIS INFO, WISE DECISIONS CAN BE MADE ABOUT YOUR DATABASE AND COLLECTIONS

DOCUMENT DB MODELING GUIDE

Analyze and Document

Here is an example of the detailed documentation produced by a MongoDB consultant as they analyzed system needs behind the writes of a single collection for a fictional nationwide coffee chain:

Attribute	Value
Description	Making a cup of coffee at rush hour
Type	Write
Frequency	3 000 000 writes/hr 833 writes/sec
Size	100 bytes
Consistency/Integrity	weak
Latency	< 10 sec
Durability	weak
Life/Duration	1 year
Security	None

These details, along with similar analysis for all the other operations and queries in the system, were used to estimate database disk and growth figures, as well as decide between replication (< 1TB) and sharding (> 1TB)

DOCUMENT DB MODELING GUIDE

Design the Model: Collections

We are not really document modeling; we are designing the schema for documents in a collection

Technically every document written to a collection can be of a different shape and not conform to any schema.

- But that is a terrible, horrible, no-good, very bad idea. Just because you can, doesn't mean you should.

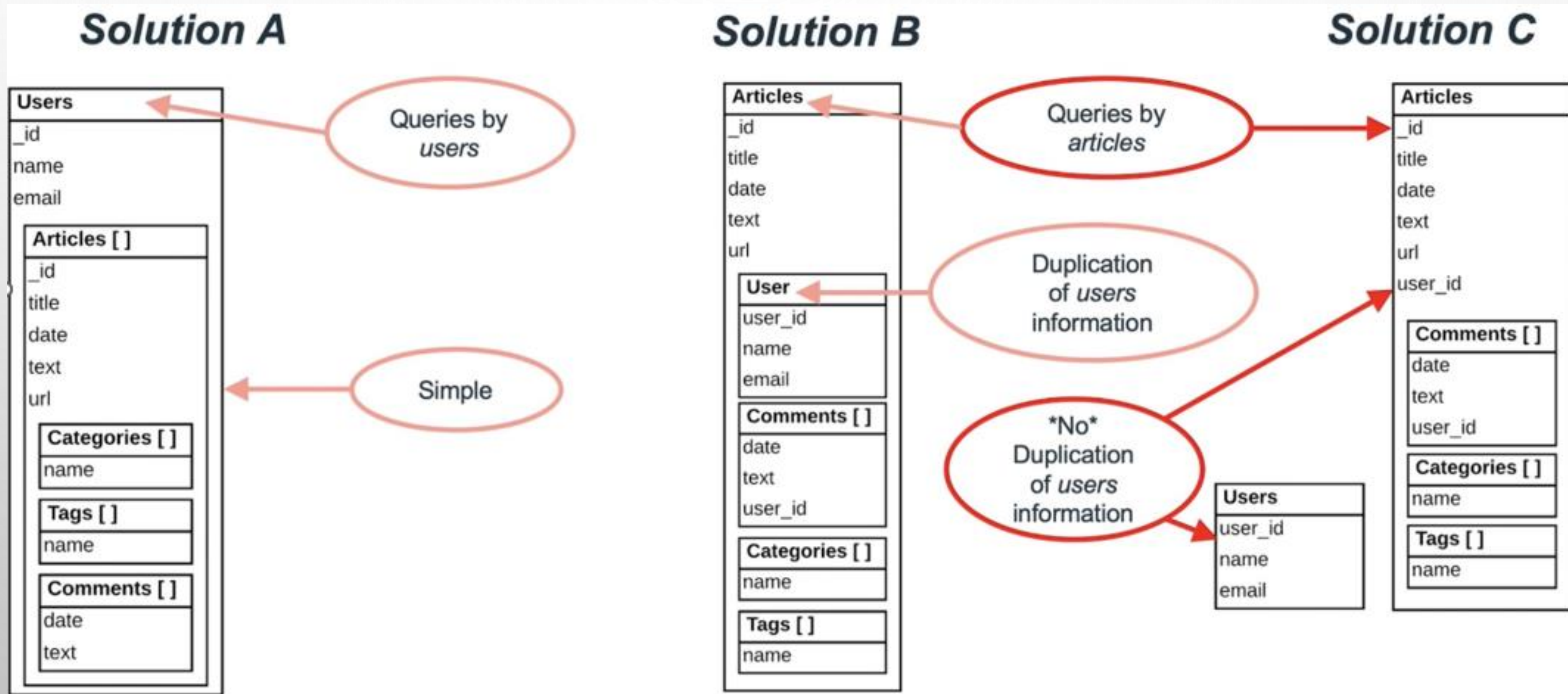
We identify and map out distinct queries and access patterns, entities, attributes, relationships and rules. This informs our first pass at designing the collections.

- Make a best guess as to which entities should become their own collection, and which should be treated as embedded subdocuments
- Implement this design as a first-level unit test, code to it, then begin iterating through user-acceptance and testing, noting areas for improvement. Repeat until you have a creamy-smooth, super-fast model

DOCUMENT DB MODELING GUIDE

Design the Model: Collections

Here is an example of iteration through three solutions until settling on one that matched system needs:



From MongoDB University training videos

DOCUMENT DB MODELING GUIDE

Design the Model: Collections

Collections are best used to represent logical groupings, critical queries, or core business entities, and are roughly equivalent to distinct tables – or more accurately, materialized views – in a relational database. So…

- Model real-world objects. Keep data together that belongs together.

For example, in a classical data modeling exercise, the instructor pulls out a business card and shows you how the info on a card can be shredded into a person, account, company, address, and contact tables. In a document database which scans and stores business cards received at a conference, you might choose to keep all data on the business card together (as subdocuments embedded in the business card document). This is an example of modeling the document to represent the real world object.

DOCUMENT DB MODELING GUIDE

Design the Model: Collections

- Better to have a large set of distinct collections, than a single collection holding the ocean.
- However, too many collections (thousands or more) creates too many namespaces deep in the WiredTiger storage engine and makes things like replication and backup/recovery take far longer than they should.
- Keeping collections dedicated to a given entity or logical grouping yields easier sharding implementation and avoids certain limitations, like the maximum of 64 indexes per collection.
- Look into Capped Collections if the app only needs to utilize the most recently inserted documents.
- If there will be millions of a given document, and it must be kept really small (one or two fields) on purpose, consider optimizing the document' s design. For example, you could abbreviate or shorten the field names. Or explicitly set the value of the _id field, perhaps to the value of a natural key in the document (keep in mind it has to be unique across documents in the collection to serve as the primary key).

DOCUMENT DB MODELING GUIDE

Design the Model: Fields

- Document databases allow a given field to be completely different from one document to another, a string in this doc, an integer in the next, and a date in the next. Or two fields in the subdocument in the first version, then four fields with a prior field now optional, then six fields and one of them is an array in the next version. Varying datatype and shape quickly becomes an unholy mess for everyone concerned. Try to ensure that a given field has the same name and datatype across the enterprise, and within the documents in a collection.
- Naming collections and fields is hard. It seems trivial, but it is hard to get right, even for modeling and engineering veterans who have been doing this for decades. If you have documented well the business purpose and meaning of the field, the name will naturally distill out of that. Follow the naming guide so that the field name is intuitive and self-explanatory. Generic names like id, text, type, name, number, etc. are problematic in the near and long term.

DOCUMENT DB MODELING GUIDE

Design the Model: Fields

Of course to make the best decisions possible about the field's design and validation options, you should have already documented the research into each field's:

- data type, optionality, cardinality, sensitivity
- default value (if any), valid values (and source of the master valid values list)
- if numeric, how many digits of precision must be supported?
- if date, should time be supported and to what level of precision?
- usage - use within queries as filter or sort predicate (informs index design)
- bounds - minimum and maximum length (if any)
- normalization and fidelity - check or change the value upon user entry to match a case or format rule, like email and SSN formatting, or validating and uppercasing street addresses vs. accepting whatever the user feeds us

DOCUMENT DB MODELING GUIDE

Design the Model: Field Datatype

- Fields can be of JSON-compatible datatypes:
    - **string, bool, object** (subdocument)**, array** and **null.**
- Since MongoDB uses BSON for document storage, it offers additional datatypes:
    - **objectId, binData, date, timestamp, double, int, decimal, long, regex, javascript, minKey,** and **maxKey.**

*Note: BSON **Timestamp** datatype is meant to be used internally by MongoDB*. Use the BSON **Date** datatype instead, for all your date and time needs.

DOCUMENT DB MODELING GUIDE

Design the Model: Surrogate & Natural Keys

Primary: In MongoDB, each document stored in a collection requires a unique _id field that acts as a primary key.

- You can manually supply the value of the _id if that makes sense for your application.
- If an inserted document omits the _id field, the MongoDB driver automatically generates it.

Natural: Let the database engine protect the integrity of the data whenever you can. Determine an identifier field, or combination of fields, that make up the natural key of the document. Create a unique index on the natural key so that duplicates are not allowed.

- This doc has more to say about unique index edge cases, like missing key field, partial indexes, unique indexes across shards, etc.

DOCUMENT DB MODELING GUIDE

Design the Model: Embedding vs Referencing

- Decisions on how to represent ownership, containment/composition and association is the most important job you have while modeling the schema for a document database. Your system should hum and function well if this is done correctly, along with thoughtful indexing, application-level joins, projections and pre-aggregations.
- The following is the single best summary I ran across on how to decide between embedding a doc, or using doc ID references. If you memorize the seven points in this image, you'll need very little of all the document modeling articles available on the internet:

Embed	Reference

• For 1:1 and 1:few relationships
• When data belongs together
• When data is read-mostly and updated rarely
	• For 1:many or 1:tons relationships
	• When data is frequently updated
	• When data is queried independently
	• When the array size is large:
	  • child referencing for 1:many
	  • Parent referencing for 1:tons

DOCUMENT DB MODELING GUIDE

Design the Model: 1:1 Relationships

- Normally straightforward and the field/subdocument is included in the collection
- There may be special scenarios where the data item is broken off into its own document
  - Example: user preferences that are typically read once and cached

```
{
_id: ObjectId
first_name: str
last_name: str
hired_date: date
manager_id: ObjectId
resume: binData
home_address: {addr_line_1: str, addr_line_2: str, city: str, state: str,
zip: str}
}
```

DOCUMENT DB MODELING GUIDE

Design the Model: 1:few and 1:handful Relationships

- Not a lot of N, but a little of N. Typically a cardinality of 1:handful will favor embedding subdocuments.
- Like a person's known email addresses, or the names of the quality inspectors who inspected a finished product. Just a few of N is a good candidate for embedding **if N doesn't need to be queried on its own, outside the context of the containing document.**

```
{
_id: ObjectId, product_id: ObjectId, product_name: str, produced_ts: date,
serial_num: str, model_num: str,
...
qa_inspectors: [ObjectId,...]
}
```

"Just because you can embed a document, doesn't mean you should embed a document." - William Zola, [MongoDB documenation]

DOCUMENT DB MODELING GUIDE

Design the Model: 1:many Relationships

- If the potential quantity of N is in 10's, to scores, to low hundreds, then it might be OK to embed in the parent document, particularly if it is just an array of doc IDs pointing to the full documents for N, or maybe sporting a denormalized field or two per reference. If you need additional information about each N in the parent doc, you use an application-level join to bring them together.
- But if the cardinality may reach several hundred to thousands or more, then it is usually better to make a separate collection for the N side, and have a parent reference on the N side, especially if each one of N carries with it additional attributes or is updated a little or a lot.
- Embedding a large array of objects is typically the wrong choice, creating bloated documents that are wrong for document databases on several fronts. Remember the 16MB limit per document if embedding large arrays of sub-documents, or arrays that can grow without bound.

DOCUMENT DB MODELING GUIDE

Design the Model: 1:tons Relationships

- "Tons" might be considered 10K, 100K or even 1M+ records that belong to the parent entity. Some MongoDB consultants and authors refer to this cardinality as 1:squillions (a word they invented). If there could be tons of N, say the historical call records for each customer in a mobile phone company, you would definitely NOT embed the call records within the customer, or even use child referencing (an array of call record document IDs kept within the customer), no matter how efficient. Can you imagine how often each customer record would have to be updated upon each new call, and the attendant index maintenance? The horror.
- With tons of N, you would typically reference the parent customer document within the child document. Using the example above, each of the millions of call record documents would contain a customer document ID pointing back to the customer that originated the call.

```
{_id: ObjectId
customer: ObjectId
called_tn: str
call_seconds: int
...}
```

DOCUMENT DB MODELING GUIDE

Design the Model: Denormalization

- Normalization is the process of breaking information down into its most atomic units and ensuring **each piece of data is mastered in only one place and not duplicated**. Each point of duplication is another chance for mistakes to be made and data to get out of sync with the master source.
- Denormalization is the process of **duplicating data in strategic places to improve performance** and optimize read operations.
- In the world of NoSQL and document DBs, denormalization is expected and acceptable. And yet the dangers posed by denormalization don't go away. Duplications must be managed by humans, code (and tests) to ensure it stays in sync with the master document or data source.
- If updates to that attribute happen frequently, then it greatly reduces the attractiveness of denormalizing.
- The more duplication, the more spinning plates must be maintained, the more likely errors will happen, the more data gets out of sync, and your company gets "never consistency."

DOCUMENT DB MODELING GUIDE

Design the Model: Denormalize from Child to Parent

- If the application demands it, you can choose the most frequently needed attributes in the child document and include them in the embedded subdocument in the parent.
  - The requirements demand <1s lookup of the customer's last 12 months of purchases. We can keep an array of purchase_id references within each customer document.
- But if the application also typically needs the purchase date and time, and purchase total, and those rarely (if ever) change, we could denormalize (duplicate) those values from the purchase collection, and include those attributes with each purchase_id in the array for each customer.
  - We would replace the array of purchase_id's with an array of subdocuments, each containing purchase_id, purchase_ts, and purchase_total values in each subdoc.
- However, just because we are fairly certain a denormalized field won't change does not mean that it won't. You must still write code to ensure that the purchase_total and purchase_ts are updated, if they ever change in the master purchase document.
- Denormalization is only a good idea if the ratio of reads far outweighs the frequency of updates.

DOCUMENT DB MODELING GUIDE

Design the Model: Denormalize from Parent to Child

- This modeling option is rarely a good idea, unless you are certain the field being duplicated from the parent to the child document will never change, or if the quantity of documents in the child collection is small. If the field in the parent source document ever does change, you may have to update every child document where that value is found, which can be a very expensive operation if the child collection is in the millions or billions. {story}
- One example where putting parent fields on the child makes sense is when the child document is an historical record of how things stood at the point in time of document creation, like an event, log, or IoT sensor reading. Historical records are rarely, if ever, touched after their initial write, so this would be a good candidate for denormalization so the various parents don't have to be joined at read-time. It is also the best use case I know of where a document DB is the perfect solution.

DOCUMENT DB MODELING GUIDE

Design the Model: Bi-directional Relationships

- This is a form of denormalizing that places references to the child in the parent document, **and** a reference to the parent document in the child. You can take it a step further by including critical fields from the referenced document so the reference doesn’t have to be joined until a less-frequently needed field is required.
- This is known as a circular relationship in relational modeling and is an abomination. But it is do-able in document schema modeling if the needs of the application demand it.

DOCUMENT DB MODELING GUIDE

Design the Model: Many:Many Relationships

- Unlike relational, an associative entity is not required to model N:N relationships. You can embed an array of references (and denormalized fields) for one entity in the other and vice versa.

```
// customer document schema
{
_id: ObjectId,
customer_account_num: str,
customer_first_name: str,
customer_last_name: str,
...
wishlist: [{product_id: ObjectId, product_name: str, added_to_wishlist_ts:
date},
...]
}
```

DOCUMENT DB MODELING GUIDE

Homework: Common Design Patterns

Use Case Categories

Patterns	Catalog	Content Management	Internet of Things	Mobile	Personalization	Real-Time Analytics	Single View
Approximation	✓		✓	✓		✓	
Attribute	✓	✓					✓
Bucket			✓			✓	
Computed	✓		✓	✓	✓	✓	✓
Document Versioning	✓	✓			✓		✓
Extended Reference	✓			✓		✓	
Outlier			✓	✓	✓		
Preallocated			✓			✓	
Polymorphic	✓	✓		✓			✓
Schema Versioning	✓	✓	✓	✓	✓	✓	✓
Subset	✓	✓		✓	✓		
Tree and Graph	✓	✓					

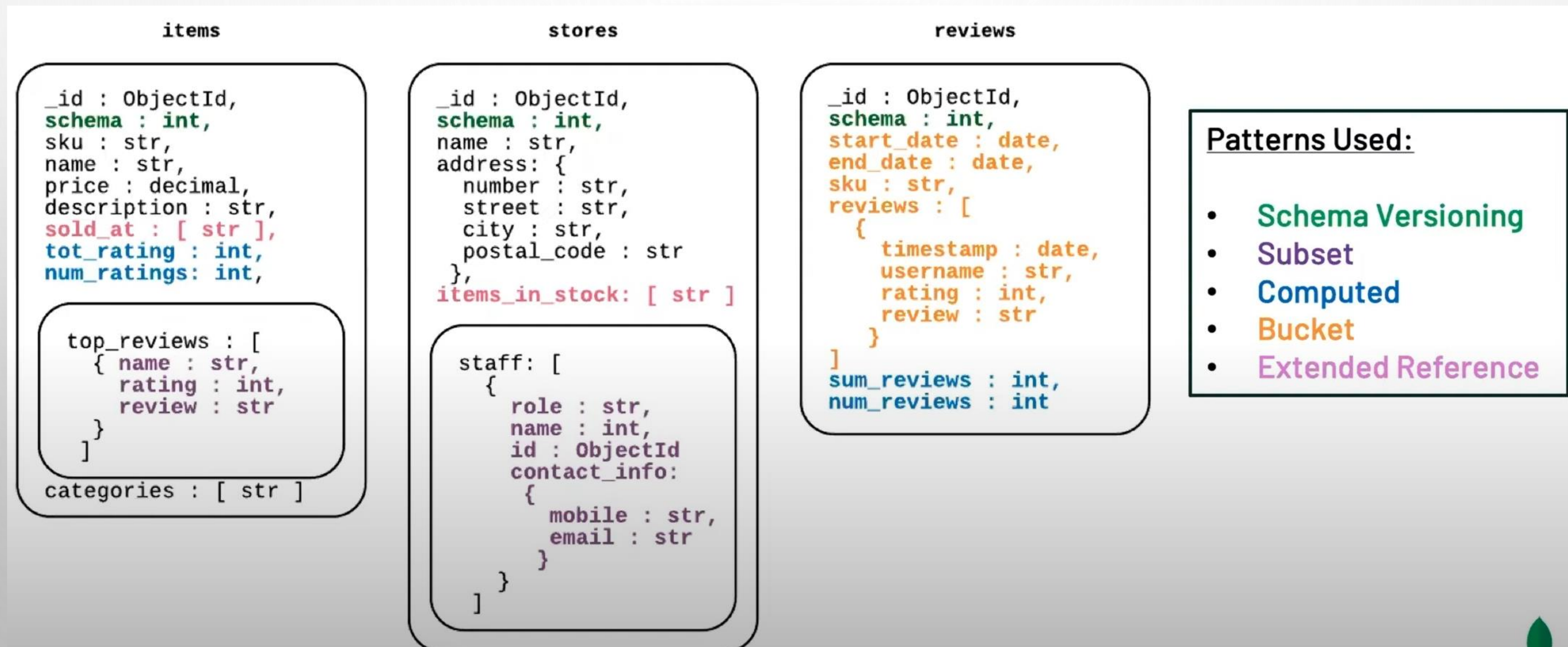
Approximation: Fewer writes and calculations by saving only approximate values.
Attribute: On large documents, index and query only on a subset of fields.
Bucket: For streaming data or IoT applications, bucket values to reduce the number of documents. Pre-aggregation (sum, mean) simplifies data access.
Computed: Avoids repeated computations on reads by doing them at writes or at regular intervals.
Document Versioning: Allows different versions of documents to coexist.
Extended Reference: Avoid lots of joins by embedding only frequently accessed fields.
Outlier: Data model and queries are designed for typical use cases, and not influenced by outliers.
Pre-Allocation: Reduce memory reallocation and improve performance when document structure is known in advance.
Polymorphic: Useful when documents are similar but don't have the same structure.
Schema Versioning: Useful when schema evolves during the application's lifetime. Avoids downtime and technical debt.
Subset: Useful when only some data is used by application. Smaller dataset will fit into RAM and improve performance.
Tree: Suited for hierarchical data. Application needs to manage updates to the graph.

Matrix and summarization from MongoDB training led by Daniel Coupal.

DOCUMENT DB MODELING GUIDE

Homework: A 10-minute Modeling Example

https://youtu.be/3GHZd0zv170?t=1550 (video timepoint goes to example of modeling from requirements for a fictional company)



DOCUMENT DB MODELING GUIDE

Protect the Model

Much like strong typing and constraints in relational databases, document databases provide features that protect the data from missing/garbage/wild values.

Use JSON Schema to validate field type and valid values.

One can add JSON Schema validation to new and existing collections:

- new collections: db.createCollection() with validator option
- existing collections: db.runCommand(callMod command with validator option)

Note: existing documents do not undergo validation checks until modification.

- validationLevel of the validator should be strict. The use of moderate is only necessary when adding to existing document databases with problems.
- validationAction of the validator should be error. The use of warn, allowing non-conforming data into the database, must be well-justified to pass architect review.

DOCUMENT DB MODELING GUIDE

Implement the Model

- Use a tool like Hackolade, DbSchema or Moon Modeler to design the schema of your collections
- Save the model to your project's github repo, perhaps /db/model folder
- Generate the collection creation statements with JSON validators and save script to github repo, perhaps /db/migrate folder
- Implement a CI/CD tool to automatically see the new file, wait for approvals, run tests with seed data, and migrate the file up the lanes, eventually to production

DOCUMENT DB MODELING GUIDE

Version the Model

- As the system requirements change, you will receive tasks to change the data model
- Open the model file in your modeling tool and update the schema design, updating the number in the schema_version field.
- Save the updated model back to /db/model. Can now see all changes to the data model over time, who did them, why and which change tickets the modification was tied to.
- The code that writes the documents will also need to be modified to accommodate the changes

DOCUMENT DB MODELING GUIDE

Homework: Model an Invoice

Valvoline Instant Oil Change

Service You Can See. Experts You Can Trust.℠

Customer Rated 4.6 out of 5*

Invoice: 1035 2/17/23, 4:49 PM R

GUEST INFORMATION	VEHICLE INFORMATION	SERVICE CENTER INFORMATION
William Coulam	VIN: UT 2006 Mazda 5 Sport, 4CYL 2.3L, MFI LAST VISIT MILEAGE: 0 CURRENT MILEAGE: 199,595	VALVOLINE INSTANT OIL CHANGE GT0030-Cedar Hills MANAGER: Sheldon 10012 N 4800 W Cedar Hills, UT 84062 385-595-4567

MAINTENANCE CHECKS

Check	Result
Lubrication Points	Sealed
Oil Drain Plug & Gasket	Checked-OK
Tire Pressure	Checked-OK Front 34 Rear 34
Brake Fluid Level	Checked-OK
Power Steering Fluid Level	Checked-OK
Battery	Tested
Windshield Wash Fluid Level	Added
Coolant Reservoir Level	Checked-OK
Transmission Fluid Level	Checked-OK
Oil Service Indicator Light	Not Reset

SERVICES PERFORMED

ITEM DESCRIPTION	QTY	UNIT	AMOUNT($)
Full Synthetic Oil Change			94.99
Valvoline 5W20 Full Synthetic Oil API SP	4.50	QT	
Valvoline Oil Filter VO80	1.00	EA	
Discount (GTKK50)			-47.50

YOUR SERVICE TEAM: CSR: Seth TOPSIDE: Seth BOTTOMSIDE: Harrison

COMMENTS

ENGINE OIL LEAK NOTED
SKID PLATE DAMAGED/LOOSE UPON ARRIVAL
THANK YOU FOR CHOOSING US
HAVE A NICE DAY!
REMEMBER US FOR YOUR NEXT EMISSION/SMOG TEST!

CHASE VISA *2141, Card Entry: Contactless, Mode: , CVM: Verified by Signature, AID: A0000000031010

Subtotal		47.49
Tax		3.44
Total		50.93
VI	(*2141 AP=07606D)	50.93
Change Due		0.00

Sign(x) ______________

Cardholder agrees to pay to issuer total charges per the agreement between the cardholder and issuer.

Save up to $7 on your next oil change

Go to www.tellvalvoline.com and tell us about your visit

Entry Code:0483 6072 0003 0035 1

Thank you for your business

INV-1: VersionAudit

Grab an invoice from your file cabinet, and imagine you've been tasked with modeling a document database with a handful of collections to tame and corral the data on the invoice.

DOCUMENT DB ENGINEERING GUIDE

Transactions and Atomicity

- If the rule of thumb in the Document section above has been followed, and each document contains all the data it will need (no lookups required), then Mongo is able to easily comply with industry-standard ACID transactions and ensure the whole CRUD operation is completed in one go, or not at all.
- As of version 4.0, MongoDB supports multi-document transactions on replica sets.
- As of version 4.2 MondoDB supports distributed transactions, which adds support for multi-document transactions on sharded clusters. If you use this feature, test assumptions thoroughly, because new offerings are typically not 100% bug-free until two major releases later.

DOCUMENT DB ENGINEERING GUIDE

Views

Read-only views in MongoDB were introduced with version 3.4.

DBAs can define non-materialized views that expose only a subset of data from an underlying collection, i.e. a view that filters out entire documents or specific fields, such as Personally Identifiable Information (PII) from sales data or health records.

As a result, risks of data exposure are dramatically reduced. DBAs can define a view of a collection that's generated from an aggregation over another collection(s) or view.

https://www.linkedin.com/in/billcoulam/https://www.linkedin.com/in/billcoulam/

DOCUMENT DB ENGINEERING GUIDE

Indexes

- MongoDB auto-creates a unique index on the _id field
- Additional indexes should be designed and created to support fields that are frequently queried, or used for filtering and sorting
- Limit of 64 indexes per collection (but once you approach 15 to 20 indexes, performance degradation becomes noticeable)
- Remember: The leading field in an index can satisfy queries that require that field, as well as the other fields in the index. For example:
- `db.orders.createIndex( {“order_id": 1, "user_id": 1, "created_at": 1} )`

can support queries that filter on

```
order_id
order_id + user_id
order_id + user_id + created_at
```

but cannot support queries that filter on

```
user_id
created_at
user_id + created_at
```

DOCUMENT DB ENGINEERING GUIDE

Indexes

MongoDB supports indexes of the following type:

- default _id index: unique (will auto-create if you forget to include it in the collection creation statement)
- **single field** index: simple index on one field. Can be ascending or descending.
- **compound** index: index on two or more fields
- **multi-key** index: automatically indexes the contents of arrays, if one of the fields being indexed is an array
- **geospatial** index: for coordinates. Two types: 2d indexes that uses planar geometry when returning results and 2d sphere indexes that use spherical geometry to return results.
- **text** index: like full text search engines, does not store stop words. Stems and stores root words.
- **hashed** index: used to index the hashed value of a field, to support sharding

DOCUMENT DB ENGINEERING GUIDE

Indexes

The indexes may utilize one of the following index properties:

- **partial** index: documents included in the index only when certain conditions are met
- **sparse** index: documents included in the index only if the indexed field is present in the document
- **TTL** index: time-to-live, special indexes that automatically remove documents from a collection after a certain amount of time

* Keep in mind that just like relational indexes, each additional index slows down document write performance a little more, and takes space on disk and in memory. Adjust capacity plans accordingly.

Note: If the query is case-insensitive, the index must support case-insensitive queries. Both the index and the query must use the same locale and strength parameters of the collation property.

MONGODB TIPS

- Mongo is not a JSON database. **Mongo is a BSON database**, with ObjectIds, native date objects, more numeric types, geographic primitives, and an efficient binary type that JSON does not support.
  - Storage is in BSON
  - Queries are BSON
  - Results are BSON
  - Even the wire protocol is BSON
  - …in other words, get to know how BSON differs from JSON to understand how to work well with it
- Unless you are reaching BigData volume/velocity, **slow queries in Mongo are typically due to missing indexes**.
  - Just like relational, additional indexes cause slower writes and updates due to index maintenance, so weigh additional indexes against response time goals
- If the application is read-mostly, consider **replication as a means of reaching response-time goals**
- **Harden MongoDB, eliminating attack vectors**:
  - Install MongoDB with the {company}-preferred method of authentication (default install used to require none)
  - Set javascriptEnabled:false in the config file
  - {Consult with DevOps/CloudOps; they usually know how to do this}

MONGODB TIPS

- Ensure **collation** is set to accent-insensitive and case-insensitive (for sorting and searching)
- **Avoid large documents, deep nesting and large arrays**
  - Try to keep documents under 1MB, nesting no more than 2 levels below the parent document
  - Arrays need to be reasonably simple or they quickly create friction (become too complex to grasp or code to, and use without error)
  - If the array is going to be amended or re-arranged frequently, re-design to not use an array, or consider a relational DB instead
- Relational databases have an optimizer. With Mongo, **you are the query optimizer**.
  - There is a rudimentary explain() that can show if indexes are used or not.
  - Learn how to match, project, join and sort in the most optimal manner for MongoDB aggregations to run quickly.
  - Ensure that sorting is the last step. Ensure there is an index to support the sort.
    - Mongo will throw an error when the 100MB memory limit for sorting is exceeded. If this is commonplace, use allowDiskUse option to utilize temporary files for sorting and pipeline aggregation.
  - Ensure there is an index on any reference columns (document ID referring to another document, aka lookups)
  - Ensure production code that uses $limit() uses $sort() first

MONGODB TIPS

- Set **journal.enabled** in the config file and ensure the commit interval matches the business' tolerance for data loss
- Use the multi:true option in document.update() to **ensure all documents that match the criteria are updated**. {VERIFY. This was true in 2018; prefer document.updateMany() now}
  - The “multi” parameter in update() defaults to false and therefore only one document will be updated.
- **Key order** in a Mongo document matters.
  - Key order doesn’t matter in JSON, but Mongo uses BSON under the covers, where order does matter, e.g. { a: 1, b: 2 } does not match { b: 2, a: 1 }.
- Be very careful with and do your **research on null and undefined** within the latest Mongo documentation.
  - Prefer $exists over requiring the presence of a field with a null value for backward compatibility
  - { $eq: true} is not the same thing as {$ne: false}
- Use db.setProfilingLevel(1, 100); to enable slow query logging (100 means queries that exceed 100ms; adjust this threshold as you see fit). {VERIFY. A 2018 doc says this is now enabled by default.}

MONGODB TIPS

- Use encryption to protect sensitive data at rest or in transit
  - Mongo features client-side field-level encryption (FLE) so data can be encrypted in-transit
  - Mongo features server-side encryption so data is encrypted at rest.
- Mongo features change streams
- Become familiar with the concepts and options in these three documents: aggregation framework, read concerns, and write concerns.
- Use Atlas to identify problem areas and hotspots
  - Navigate to the Performance Advisor (available in M10 clusters and above) or the Data Explorer (available in all clusters) and look for the Schema Anti-Patterns panel. Use it to identify and fix design errors.
  - Use Query Executor and Query Targeting tabs to pinpoint inefficient queries
- Mongo skip() and take() commands can be pretty slow. Used to be called out by Mongo themselves in the cursor.skip() docs. May need a custom query using an index, range query, sort and limit to be fast.

MONGODB TIPS

- Mongo sequential query execution really trips over itself, especially in systems where data ingest and data access are both heavy and concurrently demanding
- Index creation on large collections can bog down response time on read/write requests to that collection. Plan accordingly to do **index creation during off-peak hours.**
- Avoid diving into and using aggregation pipelines until you have a solid grasp of the basics (and you actually need aggregation pipelines)
- **Mongo requires the working set to fit in memory**
    - working set = the amount of memory needed to hold the collections and their indexes when running in a steady state at peak
    - do the necessary analysis to determine the size of tables, indexes, and memory required for the working set <u>before</u> database creation or sharding decisions are made
    - use TTL and partial indexes when you can, because they use less space on disk and in memory

MONGODB TIPS

- Offset TTL indexes by half a day, which may allow the TTL operations to happen during off-peak periods
- Use Atlas rolling index builds (when building an index across a massive sharded cluster)
- Ensure that a new application release, which introduces a new or changed query or data shape, is supported by an index
- If this system will require massive scaling, don’t just separate concerns into separate collections, separate them into separate clusters.
- Replicate Mongo data to another datastore to offload read-intensive use cases, like inverted indexes, columnar indexes, fraud and anomaly detection, ML predictions, search, leaderboard, etc. Could tail Mongo oplog, custom or COTS pipeline, or new Change Streams to replicate data to Redis, Elasticsearch, Snowflake or a data lake.
    - Replicating entails additional personnel and moving pieces to create the pipelines, monitor and maintain.
    - Avoid using the arbiter option as a replica node.

DOCUMENT DB ENGINEERING GUIDE

Sharding

MongoDB supports 3 sharding strategies for distributing data across sharded clusters:

- hashed sharding: it involves computing a hash of the shard key field's value. Each chunk is then assigned a range based on the hashed shard key values. MongoDB automatically computes the hashes when resolving queries using hashed indexes. Applications do not need to compute hashes.
- ranged sharding: it involves dividing data into ranges based on the shard key values. Each chunk is then assigned a range based on the shard key values. If you know your application will often be doing range queries on the indexed shard key field, then ranged sharding is the right choice.
- zone sharding (previously known as tag-aware): in sharded clusters, you can create zones that represent a group of shards and associate one or more ranges of shard key values to that zone. MongoDB routes reads and writes that fall into a zone range only to those shards inside of the zone

DOCUMENT DB ENGINEERING GUIDE

Sharding Tips

- Shard your document DB if:
  - The application is read-mostly and indexes no longer fit in allocated host memory
  - The application is write-mostly and writes must be spread out across multiple nodes, which also reduces the frequency of global read/write locks
- Pick the shard key wisely.
  - **Must** know the data and anticipated data access patterns. Dev must collaborate with DevOps to make the best choice for shard key and shard key type.
- Do not shard prematurely because…
  - Sharding requires lots of hardware. A cluster (replica set) is composed of a minimum of three servers (master and two secondary). Sharding requires a minimum of six servers, plus two more for *mongos* instances (which route, balance and proxy requests for the sharded server). Add three more for each additional shard. Plan and budget accordingly.
  - Sharding makes data management difficult and queries more limited.

DOCUMENT DB ENGINEERING GUIDE

The Shard Key

The shard key should allow for the most even distribution of data across shards, while at the same time supporting common query patterns. Things to avoid:

- If the distinct values of a shard key are low, very little horizontal partitioning will be possible.
- If the frequency of distinct values is skewed, this can cause bottlenecks on “hot” shards where a key value frequency is high, and potentially unsplittable chunks.
- If the key is an incrementing value, like a numeric ID column fed by a sequence generator, all writes will go to one shard, eliminating the benefit of sharding, but incurring all the costs.
- If you are using a TTL index on the key, it creates a trail of empty chunks and all activity will be pointed at the shard holding the most recent data (this is bad).
- Queries that do not include the shard key, which forces “scatter-gather” queries, where *mongos* broadcasts the query to all shards

There are special considerations when using the _id field as the shard key, or if there are unique indexes on the collection. See this doc for further details.

DOCUMENT DB ENGINEERING GUIDE

Chunk Sizing

- The cluster-wide setting for max chunk size should also be chosen with care. Too big can cause issues (heavy system resource consumption during chunk moves, or total inability to be moved by the autobalancer).
- If the shard key is evenly distributed across small to medium-sized chunks, autosplitting and autobalancing should work well. Having too many routers (*mongos* instances > 5), or restarting the routers frequently (which then lose track of chunk sizes they were monitoring), can prevent autosplitting from happening.
- If existing chunks are not evenly distributed (skewed, some holding far more data than others), look at explicitly calling splitChunk and mergeChunk. splitChunk cuts a chunk in half. mergeChunk can glue together chunks holding contiguous ranges of shard key values. Merging might be good if the shard key or sharding strategy was poor, resulting in emaciated chunks, or if updates and deletes cause certain chunks to shrink in content. Mongo does not currently support auto-merging.
- It is possible to automate the crawling and monitoring of chunk sizes, splitting and merging, to keep chunk size evenly spread out. It is possible to enforce chunk size per collection.

TOOLS

- mtools mlogvis - to visualize slow queries
  - pip3 install --user mtools[all]
  - includes mlaunch, which aids setting up complex test environments on a local machine
- http://debezium.io/docs/connectors/mongodb/ - stream Mongo data to consumers
- Studio 3T GUI IDE for MongoDB
- Keyhole - tool to gather stats and measure performance of MongoDB instances
- POCDriver - testing tool to simulate different workloads
- jq - command line tool for JSON processing, useful for parsing JSON results returned from Mongo queries

Modeling

- DbSchema
- Hackolade
- Moon Modeler

TRAINING & COMMUNITY

- MongoDB documentation
- MongoDB Issue Tracker and Feature Requests
- MongoDB Blog
- MongoDB Developer Blog
- MongoDB Patterns
- Community Forum
- MongoDB University

Data Modeling Guidelines for NoSQL JSON Databases (from the MapR team)

NoSQL Data Modeling Techniques (from the Highly Scalable Blog)

MongoDB Schema Design docs hub

MongoDB University: M320 – MongoDB Data Modeling

GROUP DISCUSSION

- IS NOSQL THE NEW REALITY? ARE YOU TAKING THE RED PILL BY LEARNING DOCUMENT DATABASES, JSON, AND DOCUMENT MODELING, WAKING UP FROM YOU RELATIONAL STUPOR? IMO: HARD NO
- QUESTIONS?
- WHAT HAS BEEN YOUR EXPERIENCE?
  - RELATIONAL
  - RELATIONAL WITH DOCUMENT COLUMNS
  - DOCUMENT (TRUE FLEXIBLE SCHEMA)
  - DOCUMENT MADE TO LOOK RELATIONAL, HEAVY RELIANCE ON REFERENCES TO LOOKUPS

CONTACT:

- https://dbsherpa.com (BlueHost's subscribe button is missing, but leave a comment and I'll get it)
- bcoulam@yahoo.com, bill.coulam@dbsherpa.com
- https://linkedin.com/in/billcoulam