Database Systems:

- What is a database?
    - A database is an organized collection of structured information, or data, typically stored electronically in a computer system
    - A database is typically controlled by something called a database management system (DBMS)
- DBMS
    - What do DBMSs provide beyond “just files”
        - Security
            - Authentication
            - Authorization
        - Schema modifications to the underlying data store
        - Isolation between multiple processes
        - Multi-user environment that allows parallel data access and data manipulation
- History
    - 1960s - the first databases began to appear
    - Databases evolved from flat files
    - IBM SABRE system for airlines
        - Helped american airlines handle reservations
    - 1970s - E.F. Cobb proposes the relational database model
    - Ingres
        - QUEL query language
    - System R
        - SEQUEL query language
    - E/R diagrams are proposed
    - Application designers could abstract away from table design
        - Is this a good thing?
    - 1980s to early 1990s:
        - The golden age of databases
    - SQL becomes a standard
    - Many new players
    - Lots of money was made
    - Early 1990s: big shakeout
    - Many vendors went bankrupt or became bit players
    - Even Oracle came close to bankruptcy in 1990
        - Accounting shenanigans
        - “An incredible business mistake”
    - Late 1990s: internet boom
    - A resurgence in vendors, Oracle and DB2 dominant
    - Microsoft started making inroads with a fork of Sybase called SQLServer
    - Today these are the three dominant commercial players
    - Late 1990s: open source databases
    - 1995 MySQL

        - Bought by Oracle
    - 1996 Postgresql
        - Based on Ingres at Berkeley
    - 2000 SQLite
        - Super lightweight database that we will use
    - Post 2000 - maturity
    - Three major commercial vendors
    - Two major open source options
    - Rise of NoSQL
        - We look at two
            - MongoDB
            - Redis
    - Today
        - Rankings are pretty stable
        - Oracle continues to lead
        - SQLite is becoming more popular
- What is a database?
    - A system that stores and retrieves data
    - A query processor
    - Some sort of data definition language (DDL)
    - Typically provides network access
        - Not always! SQLite doesn't
    - Makes some guarantees about data
- Sidebar: why a cylinder?
    - Probably due to the original appearance of hard disks
- Anyway, so, back to data
    - DBMS guarantees = transactions (mostly)
    - ACID
        - Atomicitiy - all completes or all fails
        - Consistency - data consistency constraints are enforced
        - Isolation - transactions complete as if no other transaction occurred
        - Durability - data won't be lost
    - ACID, who needs it?
        - NoSQL stores often relax or more of these constraints for performance reasons
        - Very popular in the early 2010s!
        - Since the industry has shown some regrets
        - A return to standard SQL systems or mixed systems
- Designing a proto database
    - Let's open up a good spreadsheet and make a little student database
    - Let's look at the chinook db in IntelliJ

The relational model:

- A relation is a set of tuples that follow a relational schema
- Sounds fancy, but think of tuples as rows

- Think of the schema as the columns
- The domain of an attribute is the set of all possible values for that attribute
- Maybe easier to think of a relation as a spreadsheet, but with typed and named columns rather than free-form cells

Null Values:

- Null values are controversial in both the relational model world as well as the programming world
- Is NULL part of the domain of values?
- Java primitives: No! Java Objects: Yes!
- Most databases allow null values for a column, even if that value maps to a primitive
    - What to do?
- “I call it my billion-dollar mistake. It was the invention of the null reference in 1965”
    - Tony Hoare
- SELECT email FROM accounts WHERE balance > 100000
- Some SQL to find the big rollers in your online casino to give them a nice, fat coupon
    - Don’t worry about the details right now
- What about accounts with NULL for balance?
- This is how you lose money with nulls :)

Constraints with the relational model:

- Recall that the set of constraints in the E/R diagram world was very limited
- The relational model formalizes constraints to a much larger extent
- We’ll begin by looking at keys
    - Super key - any contribution of attributes for which two distinct tuples will have distinct values
    - Minimal super key - a super key from which no attributes can be removed and still be a super key
    - Candidate key - a minimal super key
    - Primary key - the candidate key chose as the official key for the table
- What about references to keys in another table?
    - A reference to another table is known as a foreign key
- This is known as referential integrity
- Referential integrity constraints are placed on a relational model (and on database tables) to ensure that foreign keys point to actual tuples or rows in other relations or tables
- In DBMS systems this are often referred to as foreign key constraints
- In general, these are expensive
- Some systems drop them to make insert performance faster
    - Ruby on rails doesn’t use them for example
    - Seems crazy, but I have rarely seen an issue without them
    - Please don’t tell database administrators about this!
- Additional constraints
    - A string must be a valid email
        - Data type?
        - Regex?

        - A string must be from a set of choices
            - Aka an enum
        - A column must be greater than or equal to another column
- The relational model and practical databases are often not good at expressing these additional constraints
- In those situations, these types of constraints are typically enforced at the application level (that is, in your code)
    - What's the problem with this?
- Suppose your database doesn't support an "email" type. How do you verify it?
- Regex?

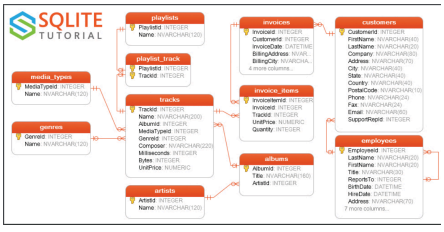
Keys:

- In the real world, almost all database tables will have what is known as a synthetic key
- A synthetic key is a key that is usually auto-generated by the DBMS for a relation when it is inserted
- Very convenient
- Problem: when I insert a new row in a DBMS, how do I know what the generated key was?!
    - Recall UUIDs: one nice feature of them is that you can generate them "up front" without relying on the database to do it for you

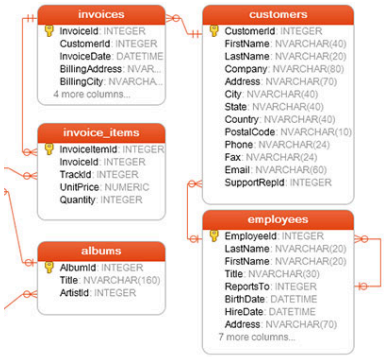
Operations on relations:

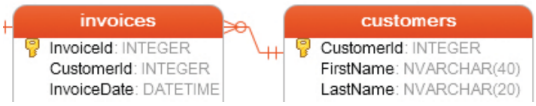
- Relational systems obviously have retrievals: reading tuples from the relation and potentially transforming and filtering them
    - SELECT is the SQL syntax to retrieve data
    - A large part of this class is going to be teaching you to be a SELECT ninja
- But you also have update operations
    - insert/create
    - Update
    - Delete
- Also associated with these operations is the notion of a transaction
    - An atomic unit of work against the database
- You will hear these operations referred to as CRUD
    - C - create
    - R - read
    - U - update
    - D - delete
- Many real-world problems involve basic digital system sitting in front of a relational database
- The value is in the relations, not in fancy logic or code around those relations
- Today, these digital systems are often web applications
    - Oh, that's just a CRUD front end on a database
- We will be building just such a CRUD app in our class
- Based on the chinook database schema
- Let's take a look at it

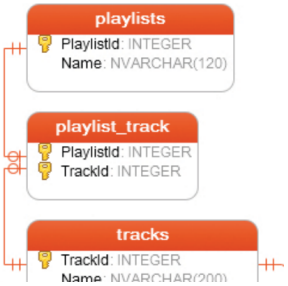
The relational model:



- Notice the universality of Integer ids
- Notice the foreign key references
    - invoices.customerID
    - This is how relationships are encoded in the tables
- Notice that employees has a self-referential key
    - This encodes a tree
- 1-N relationship
    - FK is in N table
- N-N relationship
    - Done with a join table with FK of both tables
- 1-1 relationship
    - FK can be in either table




Database Normalization:

- Designing a proper database
- Structuring database tables such that
    - Redundancy is minimized
    - Data integrity is maximized
- Edgar F Codd: a pioneer in databases
    - Proposed “1st Normal Form” in 1970
    - Went on to propose many more increasingly strict normalized forms
- Most famous Normal Form is BCNF: Boyce-Codd Normal Form
- We will be focusing on
    - 1st normal form
    - 2nd normal form
    - 3rd normal form and BCNF
- BCNF and 3NF are equivalent in the presence of a single column key
- What are keys?
    - A key is a set of attributes (columns) that uniquely determine a row
    - The literature uses the term superkey, which makes it sound cool
    - In a relation with no duplicates, the set of all columns is a superkey
    - A candidate key is a key from which no attributes can be removed without causing it to no longer be a key
    - The primary key is the candidate key used to identify rows in the relation

        - In practice, an ID column is typically added to a relation to be the primary key. This is sometimes called a surrogate or synthetic key.
- 1st normal form; we need to eliminate duplicate data to get into 1NF
- Most real world databases there will be a surrogate key column, making the database 1NF
- 2nd normal form
    - To achieve 2NF, all data must depend on the entire key
    - Again, this is trivially true with surrogate keys
        - You can start to see why surrogate keys became a standard
    - We remove redundancy and we make it easier to avoid update errors
- 3NF
    - Demands that all data depend only on the key
    - 3NF typically satisfies BCNF, especially with surrogate keys
    - Data redundancy has been minimized
    - Update complexity has been minimized
- Normal form summary
    - Each non-key column in a relation depends on
        - The key (1NF)
        - The whole key (2NF)
        - And nothing but the key (3NF/BCNF)
        - So help me Cobb ;)
    - In the presence of a surrogate key, things become pretty obvious
        - In industry, there is always a surrogate key
    - What's the general principle?
- Don't Repeat Yourself (DRY) !!
- Denormalizing
    - We've talked about how to normalize a database
    - Would we ever want to denormalize a database?
        - Yes!
            - Performance is the biggest reason to denormalize data
            - Say you want to find all students who didn't pass a class. The denormalized table would be faster to work with
            - No need to merge two tables together, just a simpler filter
    - Denormalization is basically caching at the database level
    - "There are only two hard things in computer science: cache invalidation and naming things" - phil karlton
    - Be careful, but judicious denormalization can be a big win!

E/R Diagrams

- E/R stands for Entity/Relationship
- A way to model data as entities and relationships
- E/R diagrams are visual representations of a data model that capture these relationships
- E/R modeling
    - An E/R model consists of three core aspects
    - Entities

        - The “nouns” of the systems
    - Attributes
        - The “primitive” properties of those entities
    - Relationships
        - The relationships between the entities in the system
- The E/R model is similar in some ways to Object Oriented programming
    - Entities roughly correspond with classes
    - Attributes roughly correspond with fields
- But it is different in important ways
    - Relationships are much fuzzier in their mapping
- Note that the E/R model has no notion of domain logic
- There are no methods on the entities
- It is a “pure” data model
- Relationships in E/R diagrams can have one of three different core cardinalities
    - 1 to 1 - there is always one entity on one side of the relationship and one on the other
    - 1 to N - one side of the relationship is always one, the other size is any number
    - N to N - there are an arbitrary number of entities on either side of the relationship
- These core relationship cardinalities can be further refined by defining the behavior around zero
    - Zero-or-one to one
    - One to zero-or-many
    - One-to-one
- It is debatable if foreign key properties belong in the E/R diagrams
    - This information is already captured in the relationship line between the entities
- In practice, they are almost always included even though they are redundant
- This shows an important distinction between the concrete database model and the abstract entity relationship model
- I tend to bias towards the concrete database model
- Where do we put a foreign key to support a many-to-many relationship
    - This shows the tension between the abstract entity relationship model and the practical database schema
    - In order to support many-to-many relationships in the real world, we need to introduce a join entity
- A join entity is an entity that sits between two other entities and provides support for the many to many relationship
- It is called a “join” entity that will become clear when we start discussing SQL
- As always, the foreign key goes on the N side of the relationship
- In this case, a foreign key to both the class entity and the student entity are added to the attempt entity
- Join entities are often a more important piece of data “trying to get out”
- Pure join entities do exist, but they are rare
- Often in a data modeling exercise what start as “just” join entities will end up becoming much more important when you finish doing the modeling

- You can put the foreign key on either side of the relationship
- Often it is put on the less important entity
- Why would you avoid putting the foreign key on both sides of the relationship?
- The E/R model captures important information about our data model
    - The entities
    - Their properties
    - The relationships between entities
- But it doesn't capture a lot of important information
- All this to say that the E/R model and E/R diagrams are limited in some important ways
- They often provide and excellent introduction to a data model
- But many important details are left out and must be found out via other information
- Industry
    - LucidChart is a popular E/R diagram tool
    - Will need an E/R tool for our first hw (assigned on Friday)
    - How often are E/R diagrams used in Industry?
        - Fairly frequently
            - More common the larger the company is
            - Sometimes there is a stigma against E/R diagrams
            - Often used at the start of a project, then abandoned as project matures

Database Design:

- We have worked with E/R diagrams
- Often a good first pass at a database design
- Makes your needs concrete
    - May expose data problems
    - May give you insights into what data is needed
    - N-to-n -> entity transformation is often valuable
- General process
    - What are the core nouns in my system?
    - What are their relationships?
        - 1-1
        - 1-n
        - N-n (hmmmm)
    - What other nouns are needed to properly model things?
        - Things I haven't thought of
        - Decomposing existing nouns
    - GOTO step 1
- Designing a database:
    - Once you have an E/R diagram, you can transform it into a relational schema
    - A relational schema typically reveals more detailed needs in your data model
    - Finally, this can be transformed into actual database definition statements
    - Yet more details and issues will come up
    - We will cover DDL later in the course
- E/R to database translation

        - Entity -> Table
        - Relation -> FK(s)
            - 1-1 - pick the “lesser” relation to store the fk on
            - 1-n - store the fk on the N side of the relation
            - N-n - is there an entity here?
                - If yes, add it
                - If no, create a join table (a table with two FKs only)
- Designing a database
    - As you begin working with the database schema, problems will inevitably come up
    - You will address them in the database directly
    - The E/R diagrams will become increasingly out of date
    - Application architects can become increasingly irrelevant if they continue to use E/R diagrams and other artifacts to discuss the system
        - They may also become increasingly annoying
    - Maintain a sense of humor during this process
- Advise on database modeling
    - As the system matures, accept that data modeling mistakes made early on are going to stick with you
    - The database schema get harder to fix over time
        - That’s OK! Learn to live with it
    - Question: why is a database so much harder to fix than code?
    - Start with E/R diagramming
        - But get too attached to it
    - Get to a real world database ASAP
    - Iterate your app and database as quickly as possible early on when your database size is relatively small
    - Try to figure out the crucial entities in your system and get those as right as possible
    - Read up on stoicism

Polymorphism and Databases:

- Coding with databases
    - Almost all databases are fronted by some software
    - That software is written in a particular language
    - That language is probably object oriented
        - Java (this class)
        - JavaScript :(
        - PHP XD
- Polymorphism
    - You are probably familiar with this idea from your object-oriented classes
        - Super-classes
        - Sub-classes
        - Sub-classes extend the super-class
            - Add methods and attributes

- The relational model
    - Relations are just tables and foreign keys
    - How should we model this object hierarchy if we want to store it in a table?
    - Three approaches
        - Single table inheritance
            - A single table is used for all sub-class instances
            - The columns are the union of all columns of subclasses
            - Advantages?
                - Simple
            - Disadvantages?
                - Redundant information, sparse data, wasteful in terms of size
        - Class table inheritance
            - There is only one table per class in the object hierarchy
            - Sub-classes include a foreign key reference to their parent classes
            - Advantages?
                - Easy to see how everything relates together, no sparse data problem, only store data that is actually needed, easier to understand
            - Disadvantages?
                - More expensive to compute, takes longer, more complicated
        - Concrete table inheritance
            - There is one table per concrete class in your object hierarchy
            - Advantages?
                - Everything is in one place, don't need to do joins, multiple table inheritance, no blank spots, no sparse data problem
            - Disadvantages?
                - You lose the polymorphism aspect at least visually, not obvious that there is any commonality between classes, have to update each class instead of one
- Polymorphism in practice
    - Most systems that I have experience with use a mix:
        - Single table inheritance for closely related things with many foreign keys in common
        - Concrete table inheritance for more distant relations
        - I have never seen class table inheritance work out well
    - My advice is to focus on foreign keys: the more keys two objects have in common, the more likely you are to prefer single table inheritance
- The object-relational impedance mismatch
    - While inheritance is one issue, there are many others:
        - Encapsulation isn't part of the relational model
        - Interfaces don't exist at the relational level
        - Field accessibility isn't specified at the relational level

        - Database transactions do not map well to objects
        - And so on
    - We will discuss this more thoroughly when we talk about object/relational mapping tools
    - There is, however, a fundamental conceptual and cultural difference between Object Oriented and Relational thinking
    - Object oriented
        - Imperative
        - Nouns and verbs
        - Graphs
    - Relational thinking
        - Declarative
        - Normalized
        - Set relationships
- Object Databases
    - Some people reject the relational model entirely
    - OODBMS (object-oriented database management systems) explicitly encode OO semantics
    - Smalltalk/GemTalk
    - Java/db4o
    - A small but passionate community

SQL Introduction:

- History and the select statement
- History:
    - Recall, developed by IBM
    - Donald Chamberlin and Raymond Boyce
    - Originally called SEQUEL but changed to SQL due to trademark issues
    - Initially every vendor had its own variant of SQL
    - Oracle SQL for example was not compatible with DB/2 SQL
    - This is still true to an extent
        - MySQL and Postgresql have different features and flavors
        - Case sensitivity is a big one!
            - MySQL is not case sensitive
    - Standardization efforts
    - SQL86, SQL89.. SQL2016
    - SQL99 is a popular standard
        - Standardized majority of the SQL language
        - Was heavily referred to during the dotCom era
        - MySQL kinda sorta implemented it
- SQL as a language
    - SQL is a declarative programming language
        - You tell the computer what you want, not how you get it
    - It's a functional language!
    - Perhaps the most successful functional language in history

        - Sorry, haskell rascals
    - Is SQL a popular programming language?
    - Not a top 5 programming language
    - Most “real” programmers don’t talk about it too much, but….
    - Lots and lots of jobs require SQL
    - So SQL is
        - A functional programming language
        - Very practical
        - Incredibly lucrative
    - Wtf
        - Crazy world, dunno what to tell you kids
    - On top of all that, SQL isn’t really that hard
    - It can get pretty crazy, for sure, but the basics aren’t bad and it tends to compose as a language pretty well
    - Let’s begin our journey into the language
    - SQL consists of a variety of statements
    - Today we will be talking about the SELECT statement
    - As you might suspect, a select statement starts with the word SELECT
    - SQL is NOT case sensitive
    - However, there are case conventions
        - Key words are all caps
        - Tables are often capitalized
        - Columns are usually lower case
            - Sometimes camel case, sometimes underscore separated
- SQL - Select
    - The SELECT statement is the most complex statement in SQL
    - As you can see to the right, mathematical statements are possible
    - The result of this select statement is the value 2
        - Earth shattering, I know
    - That’s not very interesting, let’s actually select some data from our database
    - Here you see selecting specific columns FROM a specific table
    - Returns these column values for all rows
    - Perhaps you want to select all values from a row
    - You can use the asterisk operator to return all columns
    - Pros
        - Shorter
        - Easier to get right
    - Cons
        - Might bring back unused data
        - Unclear what columns are available
- SQL - Where
    - Typically not useful to bring back all the data of a table
    - You often want to find a particular piece of data
    - Enter the WHERE clause

        - The WHERE clause allows you to give predicates that a row must satisfy in order to be included in the results
        - Show me the name of all tracks that are longer than 3 minutes
- SQL - SELECT
    - This is the general form of SELECT
    - Simple, but powerful
        - SELECT
            - Column list
        - FROM
            - Table
        - WHERE
            - Search condition
- SQL - WHERE
    - Recall foreign keys
    - There is a 1-N relationship between albums and tracks
    - The tracks table has an AlbumId column
    - We can use this column in WHERE
    - Give me the name of all the tracks on the album with the AlbumID of 1
    - Note that the = operator is a single character!
- SQL - Combining Predicates
    - You can combine predicates using the AND and OR expressions
    - Give me the name all the tracks on the album with the AlbumID of 1 that are also longer than 3 minutes
- SQL - Comparison Operators
    - Most operators will be familiar to you from other programming languages
    - The major exceptions:
        - Equals (a single = character)
            - Double equals usually works too
        - Not equals (<>)
- SQL - Logical Operators
    - We have seen AND and OR
    - There is also NOT
    - What does this query mean?
    - Careful with binding!
    - Return the name of all tracks not on album 1 and that are longer than 3 minutes
    - Use parentheses to get the right expression
    - Return the name of all tracks not on album 1 and that are not longer than 3 minutes
    - The IN operator is extremely useful
    - All rows where attribute value falls into a subset
    - Can be used with what is called a SubSelect for advanced queries (covered later)
    - The BETWEEN operator less widely used
    - Can be replaced with two comparison expressions
    - Might be clearer in some cases

        - Give me the name of all tracks between 2 and 4 minutes long
        - The LIKE operator can be used for string matching
        - Incredibly useful
        - Percent (%) is a wildcard
        - Give me the name of all tracks that start with a capital A
        - Very useful…
        - Also expensive
        - Difficult to index for in general cases
        - Google doesn't use relational databases for search, for a good reason
        - Still, if you don't have google-size data, it's great!
        - NULL checks
            - IS NULL
            - IS NOT NULL
        - What does NULL mean?
            - Depends!
    - Some SQL problems
        - "Give me all tracks on the album with ID 5"
        - "Give me all tracks where the price is less than the runtime"
        - "Give me all tracks not on album 1 but that belong to the same artist"
        - "Give me all invoice items whose total price is greater than 10"
    - SQL - The SELECT Statement
        - That's a lot of stuff but it's not too bad I hope
        - But you have probably learned about 20% of what you need to be a useful employee in many tech firms
        - Next lecture we will discuss sub-queries, which expand on this basic knowledge
        - Play around with IntelliJ
        - You can't hurt the database with a SELECT
            - Except for performance ;)

Joins

- Correlating tables with foreign keys
- Recall in the last lecture we learned about SELECT statements
- Those SELECT statements applied only to a single table
- Interesting, but limited in what can be achieved with them
- Today we are going to be talking about two mechanisms for correlating data across tables
    - Joins
    - SubSelects
- Joins are the more common mechanism for correlating data across tables
- The basic join syntax is
    - SELECT <cols>
    - FROM <table name>
    - JOIN <table name> ON
    - <condition>
    - WHERE <predicates>

- An inner join returns all rows for which the condition in the join clause is true
- Can also be written out explicitly with an INNER keyword
- Inner joins are the most common sort of join you will work with
- Note that the join condition uses the Foreign Key in tracks and the key in albums
- The join condition is usually an equality condition, but it doesn't have to be
- Natural Joins
    - It is often the case that the columns in one table line up with the columns on another table
    - FKs typically have the same name as the referred table
    - If this is the case, you can use a natural join and omit the equality expression
    - A little cleaner
    - However, using a natural join is rare in practice, why?
- Outer join
    - Inner joins are fairly intuitive
    - But sometimes you want ALL the rows of a particular table, even if there isn't a match in the joined table
    - To accomplish this you need to use an OUTER JOIN
    - Note that I am selecting values from both the artists as well as the albums table
    - This query will return all artist/album combinations as well as artists who have no albums, with the album value set to null
    - If I changed this to an inner join, artists who had no albums would be excluded from the results
    - We could change this to a RIGHT OUTER JOIN to include nulls from the right table (in this case albums)
    - The artists table is considered "left" of the albums table
    - Hence LEFT OUTER JOIN
    - And, finally, we could use a FULL OUTER JOIN to include values from both tables that do not have join matches
    - As you might have noticed, SQLite does not support RIGHT Outer Joins or Full Outer Joins
    - Left Outer Joins are not super common
    - Right and full outer joins are even rarer
    - What are some practical real world examples we can come up with when we might want an outer join?
- Self Joins
    - Chinook DB has a self-referential table in it
        - This is a common mechanism for encoding a hierarchy
    - Employees have a reference to their boss via the ReportTo foreign key
    - How do we deal with that?
    - It's pretty much the same technique using a JOIN
    - However, you need to alias the table to a new name
    - And you need to disambiguate the column names
    - Both of these use the "as" syntax in the SELECT clause
    - What error could we introduce into our hierarchy, given the current data model?

    - How would we model a graph rather than a tree?
- Joins as Venn Diagrams
    - Recall that the relational model is based on set theory
    - A more theoretical way to think about the types of joins is to use a Venn diagram
    - Inner Joins are the intersection of the two relational sets
    - Left Joins are all the left table and whatever matches in the right table
    - And full outer joins include all rows from all tables
    - I am not particularly inclined to theory, so I keep the following rule in my head:
        - Inner: ignore nulls
        - Outer: keep nulls
- More exotic joins
    - There are some more exotic joins, such as the cross join
    - This creates all possible combinations (cross product) of album rows and artist rows
    - I have never used it in the real world
    - Let's think of a possible use!
- Joining multiple tables
    - Sometimes you need to join across multiple tables to get what you want: "Give me the name of all tracks by AC/DC"
    - Tracks is separated from artists by the albums table
    - In order to "reach" across, we need to do multiple joins across both of the tables
    - The order of the join statements does not matter
    - It is conventional to link the tables across linearly where possible
    - Sometimes you end up going "both" directions: "Give me all albums with more than 10 tracks by AC/DC"
    - Column naming can get tricky when you are joining multiple tables
    - You may need to use aliasing to get exactly what you want
    - Example: both tracks and artists have the 'name' column in them
- Basic Subqueries
    - A subquery is a query inside another query
    - Allows you to base the results of one query on the results of another
    - A simple example
    - Now that we have looked at Joins, let's look at a related concept: Sub-Queries
    - A Subquery is exactly what the name suggests: a query within a query
    - Related to the concept of joins and they can often be used interchangeably
    - What if we wanted the names of tracks that are on albums whose title starts with "A"?
    - We could use the LIKE operator right?
- Subqueries
    - This seems like it should work, but it doesn't
    - Since we are using the = operator, SQLite assumes a single value is returned
    - To do what we want, we need to switch this to an IN operator
    - Now we are getting the correct answer for Give me the name of all tracks on albums that start with an "A"

- Correlated Subqueries
    - So far we have looked at subqueries that are independent of the outer query
    - The outer query depends on the inner query, but not vice versa
    - Is it possible for the inner query to depend on the outer?
    - Yes! It certainly is!
    - Here is an example
        - Note that the outer query on albums is referenced in the inner query selecting tracks
        - The sum() function is used to compute the total number of bytes of all the tracks
            - More on sum() later…
    - What is this saying?
        - Show me the titles of all albums whose tracks size sum to less than 1M bytes
    - Pretty cool!
    - But there is a catch…
    - This approach causes N + 1 queries
        - 1 query to select all album rows
        - N queries to select all tracks per album
        - Performance issue!
    - How can we get around it?
        - Denormalize the album size
        - Move condition into subquery
    - Denormalize the data into album table
        - Pros?
        - Cons?
    - Move condition into the subquery using a GROUP BY statement
        - We will discuss GROUP BY in a future lecture
    - This eliminates the correlation and makes it a 2 query job
    - Where else can we use a correlated subquery?
    - In the select criterion!
    - Note that we can refer to the synthetic column in the where clause
    - Pretty advanced stuff…
    - And cool! Now we can see the size of the album in the results..
    - Unfortunately, there are often serious performance issues with correlated subqueries
    - When to use joins vs when to use subqueries
    - The rule of thumb is to favor joins over sub-queries
        - Historically joins have been faster in most databases
    - At times it is clearer to express what you want via a sub-query
        - Try it and see if perf is good enough

Aggregation

- Summarizing data in SQL

- Last lecture we learned about JOINS, which allowed you to correlate one table with another via a condition (typically a foreign key)
- In this lecture, we are going to discuss how to “roll data up” into useful summary data, using the GROUP BY clause
- Tracks and Albums
    - We will be focusing on the following three tables
        - Tracks
        - Albums
        - Artists
- GROUP BY
    - The general form of the GROUP BY clause is:
        - SELECT columns
        - FROM table
        - GROUP BY columns
    - This query will select all data in the tracks table and, for each AlbumID, group the results into a single resulting row
    - Not a very interesting grouping, but this does return all distinct AlbumIDs in the tracks table
    - Often you will want to give a nice name to the aggregated column, by using an alias
    - This makes it easier to work with in the resulting data
    - Group by can be combined with Joins to give even better results
    - Note that we had to fully qualify tracks.AlbumID because this column is in both the tracks and albums table
    - This makes AlbumID by itself ambiguous
    - This new query now gives us something really cool: the count of the number of tracks on each album!
    - This is a legitimate report-tier query that a business might ask for
    - Other aggregation functions
        - AVG - average
        - MAX - maximum value
        - MIN - minimum value
        - SUM - summed value
    - What is the total runtime of albums?
    - What if we wanted to extend this out to find out the total runtime of music by artists?
    - Add another JOIN and update the GROUP BY clause
    - What if we want to see the number of tracks and albums as well?
    - Attempt 1, add counts of TrackID and AlbumID
    - The problem here is that when we GROUP BY, we get a row for each unique track/album/artist combination
    - There are just as many AlbumIDs as TrackIDs
    - DISTINCT to the rescue!
    - Attempt 2

        - Yeah kids, now we're cookin with gasoline
        - What if we wanted this information for all artists having more than 10 tracks?
        - We can use the HAVING clause
        - The HAVING clause is similar to the WHERE clause, but it applies to the aggregated data
        - All predicates in the HAVING clause should work only with aggregated columns
            - SQLite doesn't enforce this
            - Not sure why
        - Show me all artists who have tracks that start with an A, and the count of tracks and albums that these tracks are on, and the total runtime of those tracks
        - First attempt… nope
    - DISTINCT
        - Speaking of distinct album IDs, there is also a DISTINCT operator that can be used
        - If you were really just after distinct AlbumIDs, this would be clearer
    - So, what is GROUP BY for then?
    - The typical use case for GROUP BY is for rolling data up with Aggregate Functions
    - You might be tempted to put this in the HAVING clause, and unfortunately, SQLite allows this
        - Not quite sure what it means
    - But SQLite warns you: this should be in the WHERE clause instead
    - Note that the WHERE clause is applied first, before aggregation
    - Then the data is aggregated
    - Then the HAVING clause is applied
    - Aggregation Summary:
        - Ok, so, we looked at the GROUP BY functionality in SQL, for creating aggregate queries
        - We saw how you can join across one or more tables to generate more useful queries
        - We discussed how you can use the HAVING clause to filter your aggregated data
        - And we talked about how the WHERE clause can still be used to filter out data before aggregation occurs
        - Pretty cool stuff!

Data Mutation:

- Creating, updating, and deleting data
- CRUD
    - Recall the acronym "CRUD" from earlier lectures:
        - C - Create
        - R - Read
        - U - Update
        - D - Delete
    - We have been reading data
    - Time to learn how to C, U, and D as well
- Create

        - Creating data in SQL is done with the INSERT statement
        - Unsurprisingly, INSERT inserts data into a table
        - The basic form is INSERT INTO <table> (<col1>, <col2>, …) VALUES (<val1>, <val2>, …)
        - Note that when we insert data into the table, we auto-generated a new ArtistID
        - If you are inserting data into a database from a programming environment, you need to know the ArtistID!
        - Unfortunately, there is no SQL standard for retrieving this information
        - This is a major issue for programmers because you must write a custom integration for each DB
        - Here's how you do it in SQLite:
            - SELECT last_insert_rowid();
        - Bulk inserts are useful for inserting lots of data
        - Data imports that are executed as one-at-a-time INSERTs can be very slow
        - This insert is a single transaction
            - It either all works or all fails
        - You can also insert the values of SELECT statements
        - Columns will be matched by name
            - You can use aliasing if you need to
        - This is useful in situations when you are doing administrative work
            - Fun LeadDyno Story Time!
- Update
    - Updating data in SQL is done with the UPDATE statement
    - The basic form is UPDATE <table> SET <col1>=<val1>, <col2>=<val2>, WHERE <conditions>
    - Careful, you don't want to forget the WHERE clause!
    - Lucky for you, you have an instructor who likes you and wants you to be happy..
        - ResetDB.java
    - Bulk updates do sometimes happen
    - Often are computed updates as shown
    - The || operator means "string concatenate" in SQLite, Oracle and a few other DBs
        - Not standard
- Delete
    - DANGER WILL ROBINSON DANGER
    - Deleting data is a dangerous action
    - Can lead to referential integrity violations
    - Often better to have a boolean 'archived' column
    - General form is DELETE FROM <table> WHERE <condition>
    - Demo IJ preview functionality
- Replace/Upsert
    - This is a SQLite specific statement
    - Replaces any existing rows that violate a uniqueness constraint with the new data

        - If no rows violate a uniqueness constraint, the row is INSERT-ed instead
        - Syntax:
            - REPLACE INTO <table>
            - (<col1>, <col2>, …)
            - VALUES (<val1, val2, …)
        - This is a variation of the UPSERT pattern (aka MERGE)
        - See the Merge(SQL) page on wikipedia
        - This common operation is very common with NoSQL databases
        - What are its advantages?
        - Upserts are not standard in SQL and some databases have varying levels of support and correct behavior for it
- Data Mutation Summary:
    - Today we learned the C, U, and D of CRUD
    - Create with the INSERT statement
        - We learned how to get the generated key after an insert occurs
            - Useful for the project!
    - Update with the UPDATE statement
    - Delete, if you must, with the DELETE statement
    - We also learned about Upserts, a newer and non-standard pattern for inserting or updating data into a database

Paging and Ordering:

- Paging
    - If you have used the internet, you are familiar with the concept of paging
    - No, not that kind of paging…
    - This kind of paging
    - SQL Database typically have support for a notion of paging
    - It is not, however, part of the SQL standard
    - In SQLite there are two optional clauses in the SELECT statement:
        - LIMIT: limits the number of results
        - OFFSET: takes an offset to start at
    - Note that these can be used independently
    - LIMIT, in particular, might be used to limit a search page to only N results
    - Offset is the total offset in records, not pages
    - So in this case we are looking at page 3 of a system that is showing 10 items at a time:
        - OFFSET 0 - page 1
        - OFFSET 10 - page 2
        - OFFSET 20 - page 3
- Ordering
    - You often wish to order the data you are displaying
    - For example, maybe you allow sorting by different columns
    - This is accomplished with the ORDER BY clause in SQL
    - You can order by any column in your query
    - Note that in aggregate queries you must order by an attribute in the final results

    - Ordering - Direction
        - By default, ordering is done in ascending direction
        - If you want to order in the descending direction, you can add the DESC
        - You can also use the ASC keyword if you wish to be explicit about ascending order
    - Ordering - Multiple Columns
        - If you wish to order by multiple columns, you can use comma separated ordering specifications
        - Order by milliseconds descending, and if two tracks are equal in length order them by name, ascending
    - Ordering - Nulls
        - Our old friend NULL
        - What does NULL mean when ordering values?
        - Kinda depends…
        - By default NULL is considered “first” in SQLite
        - If you want the opposite behavior you can use the NULLS LAST qualifier in an ORDER BY clause
        - Note that NULLS LAST means literally: NULLS LAST
        - If you run a DESC order with the clause, nulls will still show up at the end of the query
        - This is SQLite specific stuff, a different DB may have dramatically different behavior
- Collation
    - Collation is a set of rules that define how data is stored, retrieved and compared in a database
    - Consider the query at right, we have a WHERE clause with a comparison in it
    - Note that the value has extra whitespace at the end
    - Executing this query will return no results because, while there is an album with the name ‘Facelift’, it will not match this query due to the additional whitespace
    - We can change this by modifying the collation of the comparison, using the collate keyword
    - By using the RTRIM (right trim) collation for this comparison, we will get a match with this query
    - Generally there is a default collation for the whole database
        - In SQLite it is case-insensitive ASCII by default
    - Depending on the database, you can modify the collation for a table, a column, and within a query
    - SQLite has an API for creating your own custom Collations
    - What’s an example collation you might want to create?
- Explain
    - Learning how a query will execute
    - As we have discussed before, SQL is a declarative rather than an imperative language
    - SQL tells the database what you want, not how to get it

        - But sometimes you want to know how the DB will get it
            - Mainly for perf reasons
        - To learn how a query is going to execute, you can use the EXPLAIN QUERY PLAN statement
        - The results for this query are interesting!
        - The DB will do a table scan
        - The DB will then create a temporary B-Tree (a data structure) for ordering the results
        - That's unfortunate that we need to create a temporary B-Tree
        - We can fix that by adding an index on that column
        - We will discuss indexes (and B-trees) in more depth later
        - For now, just assume they mean "fast access"
        - Adding an index is pretty simple
        - And now our query is more efficient!
        - No temporary B-Trees needed!
        - Awesome!
- Optimization
    - "The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming"
        - Donald Knuth
- Paging, Ordering, and Explain Summary
    - You can page data in SQLite (not SQL) using the LIMIT/OFFSET clauses
    - You can order your results with the ORDER BY clause (standard SQL)
        - Null handling can be tricky, definitely non-standard
    - If you want to see the implementation of a query, the EXPLAIN QUERY PLAN statement will tell you how SQLite is going to implement a given query
    - With that information, you can make optimization decisions

JDBC:

- Working with databases from Java
- Using Databases
    - We've learned the basics of SQL
    - But how do we do something practical with it?
    - Believe it or not, there are people who write only SQL
    - Often integrated into some sort of reporting engine
    - But most developers today program in languages like java or python
    - In order to use databases from these languages, you will need to use some sort of API
    - The java API is known as JDBC: the Java Database Connectivity API
- JDBC
    - Is a pretty ancient API
        - Many wants to be found
    - Still, in the overall scheme of things, not terrible
    - The initial entry point into JDBC is the Connection class

        - java.sql.Connection class
- Connections
    - To get a connection, you can use the java.sql.DriverManager
    - You pass in a connection string which
        - Starts with “jdbc:”
        - Then specifies a database/driver
        - Then specifies some sort of database connection
    - In the case of SQLite, the connection strings final component is just a file path
    - For networked databases, it will often be a URL + user + name + password
- Drivers
    - Internally, the JDBC system will look up a driver for the given database
    - In the case of SQLite, the driver is org.sqlite.JDBC
    - The driver must be on the classpath to use SQLite
    - Each database has its own JDBC driver
    - Drivers provide a way to map the generic API of JDBC down to the specific implementation of a particular database
    - Beware: not all drivers implement all features of JDBC!
- Connections
    - Once a connection has been established you can use various methods on the object to configure the behavior of the connection
    - The most important of these is probably when to commit, which is controlled by the get/setAutoCommit() methods
- AutoCommit
    - If AutoCommit is set to true, every SQL command executed will be treated as its own transaction & committed to the database
    - If AutoCommit is set to false, the SQL commands will need to be manually committed
    - Best practice is to set autocommit to false on connections like that
- Isolation Level
    - Another important configuration knob to know about with connections is the isolation level
    - We will discuss transactions and isolation levels in more detail later in the course
    - For now, just know that you can control the isolation level via JDBC
- Executing Statements
    - There are a few different ways to execute sql statements
    - In this example, we saw the use of a statement
    - You create a statement with the createStatement() method on a connection
    - Somewhat confusingly, a statement can be used to execute multiple SQL statements
        - I told you, the API is old and kinda gronky
    - The execute statement returns a boolean
    - We could also have an executeUpdate(), which returns the number of rows that were updated
    - This will prove useful later when we discuss Optimistic Concurrency

    - Note that executeUpdate() can also be used for other update-style SQL statements
        - DELETE - returns the total number of rows deleted
        - Insert - returns the number of rows inserted into the database
- Executing Queries
    - So that's the update side of things, what about reading data?
    - For that, statement has a method executeQuery()
    - This method returns a ResultSet object
    - Note that the ResultSet object does not implement iterable
    - Instead, it has a next() method that both advances to the next result if it exists and returns whether it exists
    - Very C-like API!
        - Old and gronky
    - The ResultSet acts like a cursor in the database, pointing to a currently active result row
    - To get the values out of the currently active row, ResultSet provides a collection of get() methods
    - These get() methods take a column name to retrieve data from
    - They are typed by the method that you call (e.g. getString()) gets the column as a string
    - What happens if we try to get the column as an integer by calling getInt() instead
    - OOOF
    - It works (at least in SQLite)
    - It returns the string column coerced to an integer
    - This is very Driver and Database specific: some might throw an exception
    - But the larger point here is that JDBC is not statically type safe
    - Types are not checked at runtime to ensure compatibility between your database schema and java code
        - O/R tools address this issue to an extent
    - Note that you can write whatever sort of query you'd like in the select, including JOINs…
    - Note that we use aliasing here to disambiguate between the track name and the artists name
    - You can also do GROUP BY, etc..
    - The expressive power of SQL makes up for the gronkiness of the JDBC API in spades
    - Ok, what if we want to parameterize our query?
    - Let's make our little command line tool take an artist name
    - Using ResultSet we would need to use string concatenation to construct the query we want
    - Fine, but what if someone types something like this:
        - AC/DC' OR artists.name LIKE '%
    - Ooops! We have accidentally given the user the ability to escape out to raw SQL

    - Here it isn't a huge deal, but this can become a very serious issue when executing updates for example
    - What if a user was able to inject a different WHERE clause when updating account balances?
    - To avoid this problem, the JDBC has a different mechanism for generating parameterized SQL: Prepared Statements
    - PreparedStatements allow you to parameterize SQL queries by putting a "?" question mark in the queries where you want to insert values
    - You create a PreparedStatement by calling prepareStatement() with the SQL you are interested in
    - This SQL has a ? in the LIKE clause
    - We can then call setString() on the prepared statement to set that query value
    - Note that we are putting the wildcard operator in the value here
        - Still a bit of risk since users can also enter wild cards
        - Could use string concatenation in the query instead:
            - '%' || ?
    - Note also that parameters are base 1 not base 0
    - The database driver then does all the escaping necessary on the value you are setting into your query
    - Dramatically reduces the attack surface area!
    - Finally we call the executeQuery() method on the PreparedStatement to get the resulting values
    - This produces a ResultSet that behaves in the normal manner
    - By and large you should be using PreparedStatements (or the equivalent in other programming languages) in your java coding
    - Often faster than statement, and always far more secure!
- Cleaning up
    - Thus far we have been using things like connections and prepared statements without cleaning up after ourselves
    - But this is java, isn't everything garbage collected?
    - Nope
    - Things like file handles, network sockets, database connections, etc… must all be manually managed
    - We need to manually call the close() method on the connection object
    - Here's a naive first shot at that
    - What could go wrong?
    - This is better, we now guarantee we will at least try to close the connection after the SQL executes
    - We really should be closing the statement too though
    - Would be annoying to have a bunch of try/catches for each thing we want to close
    - Fortunately, java has introduced syntax to help here, the try-with-resources statement

        - A bit confusing (I wish they had picked the syntax with() instead) but this will automatically close the connection and statement
- Connection pools
    - The final thing I want to discuss around JDBC isn't really part of JDBC, but is part of most practical systems: the idea of a connection pool
    - Making connections to a database can be expensive, especially if the database is across the network
    - Connection pools, as the name suggests, pool connections so that when you close one, it doesn't really close
    - Instead it is returned to the connection pool and "cleaned up"
    - HikariCP is one such connection pool
    - You instantiate a HikariDataSource and configure it with a URL and so forth, much like a normal connection
    - The DataSource will wrap actual DB connections
    - These connections will present the normal JDBC, API, but will, instead of really closing, be returned to the pool
    - Sophisticated pool management guarantees a max number of connections, etc…
    - Almost all real world systems I have worked on use some sort of connection pooling
    - Our project will not use one because SQLite doesn't really benefit from it
- Using Databases
    - Ok, that's a quick tour of JDBC
    - Lots of excruciating details, strange design choices and driver inconsistencies
    - But, despite it all, a very useful API!

Introduction to Web Programming:

- The Web
    - This week we are going to take a small detour away from databases to look at Web Programming
    - You may wonder why we need to learn web programming in a database class
    - The vast majority of web applications in the world involve some sort of database
    - It may be the case that most database applications in the world involve some sort of web application
        - Most major companies you have heard of involve both: google, facebook, instagram, etc…
    - I am going to teach you an "older" way to build web applications
    - I would call this a "traditional" web application
    - In contrast with "modern" single page application frameworks like react, vue, etc…
    - Next year I am going to create a new course CSCI 431: Advanced Web Development
    - Go beyond what is taught in CSCI 331, with a focus on hypermedia concepts
    - For now, let's go through some core concepts around traditional web applications:
        - HTTP

        - HTML
        - The “web stack” we are going to use in this class for our project
    - First things first, the web involves a network protocol, HTTP
        - HTTP: HyperText Transfer Protocol
        - A mechanism for transferring hypertext documents from a web server to web clients (typically browsers)
    - HTTP consists of requests and responses
    - Requests include things like:
        - The request type
        - The request path
        - Additional information like data submitted via a form
    - Request types:
        - GET - get a resource
        - POST - create or update a resource
        - PUT - update a resource
        - PATCH - partially update a resource
        - DELETE - remove a resource
    - Responses consist of
        - A status code
        - Metadata about the response
        - Typically some content in the “body” of the response, often in the form of an HTML document
    - HTML
        - HTML: HyperText Markup Language
        - A language for describing hypertext documents
        - Influenced by XML
            - But not XML
        - We are going to stay very basic for our HTML
            - You will not be judged on your styling
            - Avoid complex HTML nesting for layout reasons
            - Focus on the data in the project
- Our Web “Stack”
    - We are going to be working with a somewhat ancient web server setup
        - Java as our language
        - SparkJava as our web server
        - Velocity as our templates
    - You will probably not work with this professionally
    - Nonetheless this is a first web platform
- Spark Java
    - SparkJava: our web server
        - It’s written in Java
        - It’s pretty simple to work with
        - It doesn’t hide much of the HTTP/HTML loop from you
    - The core of SparkJava is mapping request URLs to response strings

        - In the example we have, we are mapping the path “/” to a string produced by the template index.vm
        - The (req, resp) -> thing is a closure, or an anonymous function
        - Req - the web request object (so you can get info about the request)
        - Resp - the web response object, so you can modify the response
        - By default in Spark, you can return a string from your path handler
        - Here we render a template, the “templates/index.vm” velocity template into a string, passing in all the employees as an argument named “employees”
        - The method get() is passed in this request handler and the path “/”
        - This tells spark, if an HTTP GET occurs to “/”, call this logic
        - This defines a route, a path to handler logic pairing, in our web application
        - A little fancy pants, but it turns out to be a nice, tight way to define routes:
            - An HTTP verb (e.g. get())
            - A path
            - A “function” to execute when that path is requested via the specified verb
        - This pattern is common in other libraries as well
            - Eg. flask in python
- Velocity templates
    - We are going to be using velocity templates
    - Velocity templates are a mature template library for java
    - Templates allow you to create dynamic string content more conveniently than concatenating strings together
    - Velocity template basics:
        - $ - refer to a variable
            - $! - null safe
        - #foreach - a loop macro
        - #if/#elseif/#lse - conditional macro
        - #parse - includes another template
    - In this example, we are iterating over all the employees we found and now rendering a row for each
    - Note that in velocity templates, you can use properties, rather than java-style getters
        - E.g. getFirstName()
- Model Classes
    - A large part of the project is going to be writing Model Objects
    - Model objects are objects that correspond to your database and that expose
        - Database fields
        - Logical operations
    - Here we are using the employee model object
- Model, View Controller
    - You may have heard the term MVC: Model, View, Controller
    - This is a common system model across many different domains, but it applies very well to web programming
    - Controller

        - Responsible for processing a user action
        - dispatches/converts that action into a request to the model
        - Relays the models response to the view
- View
    - Given a model, creates an updated user interface for the user to interact with
    - NB: this could be an update in place (as with a thick client) or a complete refresh of the UI, as with web pages
- Model
    - The model, sometimes referred to as the domain model is the representation of the underlying domain
    - In OO languages, typically represents both the data and the actions available on that data in the domain
- Model Classes
    - In our case
    - Model: Our java objects that we will create
    - View: Velocity templates
    - Controller: Controller Java files
        - Demo: step through employee request
    - Note that for our model, we have a roughly 1-1 correspondence with the underlying scheme from chinook db
    - In addition to fields that map to the database, we also have methods that allow retrieval and modification of that data
    - The model class will communicate with the database via JDBC
    - As we discussed last lecture, JDBC is the core low level API java provides for working with databases
    - The example all() static function demonstrates a basic JDBC call
        - We connect to the DB
        - Create a statement
        - Execute some SQL
        - Process the results from database rows into java model objects
        - Return them as a list for display
    - Let's implement that count feature together!
    - First thing first, we need to get the count parameter from the URL
    - We will pass it in as a query parameter
    - Query parameters, in the web sense, are those funny name/value pairs that come after the question mark in the URL
    - Can be used to pass “arguments” to a given end point
    - In spark, we can use the request.queryParams() method to get the value passed in
    - It’s a string, so let’s convert that parameter to a string…
    - And then we can update our query to use the LIMIT statement
    - Fix a few compilation errors and restart our server
    - And presto! The limit works!

        - But what did we say about concatenation and query parameters?
        - What if someone were to craft a URL like this?
        - It turns out that this will not delete all employees
        - JDBC is smart enough to only execute one SQL statement
        - If you want multiple statements, you must batch them
        - But not all SQL APIs are as smart
        - This is an example of a SQL injection attack
        - As we said in the JDBC lecture, string concatenation should never be mixed with user data
        - Instead, we need to use a PreparedStatement
        - You need to specify placeholders for the variables that will be set in the query
        - You then call the appropriate set method, with an index, to set the value
        - The index is 1 based
        - You specify placeholders for the variables that will be set in the query
        - You then call the appropriate set method, with an index, to set the value
        - The index is 1 based…
        - Fine, we've reverted to an int and used a prepared statement, but now the server isn't compiling…
        - We need to parse the string into a valid integer: Integer.parseInt()
        - And we're done!
        - We have a functioning mechanism for limiting the number of employees we show with a dynamic query, driven by a URL parameter
        - Not bad!
        - Using this, as well as some helpers, you can implement paging in the app
- Web Programming Summary:
    - The core web technologies are HTTP and HTML
    - We will be using Spark Java as our web server
        - This will be our "controller" layer
    - We will be using Velocity templates for our HTML templates
        - This will be our "view" layer
    - We will be using JDBC inside "model" classes to work with the SQLite database
        - This will be our "model" layer
    - String concatenation is bad, PreparedStatements are good!

Project Overview:

- URL layout
    - We will be using a standard URL layout
        - Based on long-standing patterns in web apps
    - This is sometimes referred to as REST-ful URLs
    - That is not correct, but be familiar with the term
    - GET & POST are two HTTP methods
    - Issuing a GET or a POST to a URL is a different sort of request
    - GET - get the resource at the URL
    - POST - update the resource at the URL

        - Consider the URL /employees/1/edit
        - If we issue a GET to that URL, we are asking for an edit UI for that resource
        - If we issue a POST to that URL, we are asking the server to update the resource
        - In HTML, a link issues a GET
        - A form can issue a GET or a POST
        - We use forms for the new employee UI and edit UI that POST back to their URL
            - What about delete? We'll talk about that in a bit
        - It turns out that there are many other HTTP methods:
            - PUT - replace the resource
            - HEAD - HTTP headers only
            - DELETE - delete the resource
            - PATCH - partially update the resource
            - OPTIONS - get communication options for the resource
        - Unfortunately, HTML as it currently exists makes it very difficult to use anything other than GET or POST
        - For deletes, I would prefer to issue this request:
            - DELETE/employees/1
        - So this is the URL pattern we are going to use for all the entities in our system
        - /tracks /artists etc…
- Controller code - create
    - Recall the Model View Controller concept
        - Model - the model classes, know how to work with the database
        - View - the velocity templates that render HTML
        - Controller - the code in the server class
    - There are two aspects of creating an entity
        - Show the create UI
        - Do the creation of the element
    - Correspondingly, we have two routes
        - One GET route
            - /employee/new
        - One POST route
            - /employee/new
- Create - GET form
    - The GET is pretty simple
        - Create a new employee object
        - Render the employees/new.vm template
    - The new template has a form in it that POST back to the same URL
    - The form body is extracted to a form.vm file so it can be shared with the edit functionality
- Create - POST
    - The POST logic is more complex
    - Note that the new template has a form in it that POSTs back to the same URL
    - This URL is overloaded with a GET and a POST operation
        - It is a resource that can be manipulated via different actions

        - Again, the form body is extracted to a form.vm so it can be shared with the edit functionality
        - On the POST, we put the values from the request into a new employee object and attempt to create it
        - create() will validate() the object and, if it is valid, create it, otherwise return false
        - On a successful create, we redirect to the URL that shows the newly created object
        - On failure, we re-render the create UI and display the errors
        - Next let's look at the update code….
- Update
    - Almost identical pattern to the CREATE logic
    - Note that the URLs are now /employee/:id/edit
    - GET to display the edit form
    - POST to update the object in the database
    - Once again, we are treating the employee as a resource
- DELETE
    - Delete logic is also simple:
        - Find the given object
        - Delete it from the DB
        - Redirect to the list of objects
    - This implementation isn't ideal
        - We are using a GET request to modify a resource
- Read
    - Reads are simple compared to Create and Update:
        - Just GETs
        - No URL reuse
    - /employees renders a list of employees
    - /employees/:id renders a specific employee
- Project Note
    - Again I want to emphasize: this is NOT a web app development class
    - I want you to be writing mostly Model/JDBC code
    - We will be using simplified HTML and template logic
        - Eg. tables for layout
    - I do want you to understand the general web patterns, however
        - Be comfortable with web terminology
        - Understand the URL patterns
        - Know what a request and response are
        - Maybe learn a bit of htmx!

Search:

- Implementing search
- One of the most common pieces of functionality to implement in web applications is search
- When you think of search, you no doubt think of duck duck go right?
- Most people think of search as textual, as with search engines

- In SQL, general text search is implemented with the LIKE clause
- Recall that wildcards are '%' in SQL, not '.*'
- Here we enclose the search string with wildcards on either side, thus implementing full text search on the same name
- This is NOT efficient
    - There is a reason search engines do not use SQL databases
- However, it is good enough for many applications
- If you are interested in efficient full text search for large data sets, you would typically use a system designed for that problem
- In java: apache lucene
- Lucene: released in 1999 by Doug Cutting
- Fifth search engine
    - Xerox PARC
    - Apple
    - Excite
- Merged into the Apache project in 2001
- Like all search algorithms, Lucene builds indexes
- An index is a data structure, typically stored in a file, that assists in retrieving data from some other source
- In the case of Lucene, the source is textual information
- Indexes are typically stored in binary formats
    - This is done for efficiency's sake
    - Here is an example of some text files of quotes, and a lucene index of files
- Segments
    - Lucene indexes are composed of multiple sub-indexes, or segments
    - Each segment is a fully independent index
    - Each can be searched separately
- Files
    - .cfs & .cfe file - internal files
    - .si - stores metadata about a segment
    - Segments_1 - stores actual index
    - Write.lock - lock file, used to prevent multiple
- Command line demo
    - Indexing
    - Searching
        - Scoring
        - Positive and negative queries
        - Stemming results
- Tokenizing
    - Tokenizing is the process of splitting a string up into tokens or individual meaningful characters
    - A big part of parsers and compilers as well
    - NB: Search engines need to tokenize and parse content online
    - You *don't* want HTML tags to be part of your index

    - Imagine trying to search for “bold” or “div”!
- Stemming
    - Note that in general text search, exact matches are typically not desirable
    - Words are “stemmed”, taking morphological variants of a root/base word and reducing them to common set of characters
    - The “Porter Algorithm” is a very old (60s) very popular stemming algorithm
    - Somewhat complicated, but there are implementations available in most programming languages
    - Newer, better stemming algorithms are also available
- Lemmatizing
    - Lemmatizing probably makes more sense to you
    - Reducing variants of a word to a “dictionary” version of the word
    - Turns out to be a bit more expensive
- PageRank
    - Our demo example of lucene uses simple text-based scoring
    - In modern search engines, additional metadata is used in order to better score online content
    - The most famous metadata for internet search results is “PageRank” the original google algorithm
    - Google used inbound links from other pages mentioning the same terms to determine how important a document was
    - Very sophisticated for the time
    - Made google a much better search engine than its main competitor “Yahoo”
    - Google has since moved beyond page rank, using techniques like artificial intelligence and CIA advice to create search results
- SEARCH
    - Most databases now include a text search extension
    - SQLite: FTS5 (full text search 5)
    - Text search is DB specific, both in configuration and in query syntax
    - We are not going to be using full text search for our project
- SEARCH Implementation
    - We are going to use the standard SQL LIKE support to implement search for things like Tracks, Artists, etc….
    - This is very much like you would do things in most web applications
    - First we need a search form
        - Note that we do not specify a URL or Action so we will get the default
            - The current page
            - GET
    - When you hit enter in this input, it will submit a GET with the parameter ‘q’ (for query)
    - In this controller, we check for this parameter and, if it is there, call search() rather than all()
    - In the model, we add a method called search() that looks a lot like all() but takes a search parameter

        - IRL you would probably combine these, but let's leave them separate for clarity
        - The search SQL isn't complete: we should be able to search on artist and album too
        - That's part of your project to fix
- Advanced Search
    - Not all searches are text searches
    - For the project we are going to implement advanced track search functionality
        - Numeric aspects
        - Relational aspects
        - Text aspects
    - This is where databases shine when compared with general text search too
    - First, we have an expanded search template
    - We have inputs for various fields that are going to make up the search
        - Some text
        - Some dropdowns for relational aspects
        - Some numbers
    - The controller logic is a bit more involved than the simple search
        - We need to pick out all the values passed in, not just the text search
        - We need to do so in a way that passes null to signal "no value"
    - The model logic is more difficult
    - We are constructing a dynamic query: the form of the query depends on what the user inputs
    - In general there are two types of queries in most web applications
    - Static queries, where the arguments may change, but the form of the query does not
    - Dynamic queries, where both the arguments and the form of the query may change
    - But Carson, I thought you said that string concatenation was a bad idea!
    - It is! When the string has unsanitized user input
    - Here we are constructing a dynamic query via string concatenation, but we are not concatenating user-entered strings
    - We must construct the query via string concatenation and keep track of the arguments that will be passed in
    - We use a list for the values
    - This allows us to use an indexed loop over the variables and ensures that everything works out
- Search Summary
    - Text search is a fundamental feature of the web
    - "Real" text search uses specialization technology
    - We will be using LIKE queries in SQL to implement text search
    - Simple text searches are often integrated directly into list views
    - With a database, you can also implement more advanced searches using fields with particular data types

DDL - Data Definition Language

- Defining tables & views
- Defining tables
    - DDL: Data Definition Language
        - A language that defines your relational schema
    - Includes syntax for creating, modifying and deleting tables, views, indexes, etc…
    - CREATE TABLE - creating tables in a database
    - Syntax:
        - CREATE TABLE <name> (
            - <col name> <col type>
        - );
- Defining columns
    - Column definitions
        - A unique name within the table
        - A type
        - Additional metadata about the column
- Column types
    - Databases typically support many types of data
        - Integers
        - Strings
        - Decimal numbers
        - Dates
        - Booleans
        - Binary (blobs)
    - Integer
        - Integer types will often specify a size (number of bytes)
            - INT
            - SMALLINT
            - BIGINT
        - In SQLite these all map to the type INTEGER
            - 64 bit (8 byte) signed int
    - String
        - Like integers, string types will often specify a size
            - CHARACTER(20)
            - VARCHAR(255)
            - NVARCHAR(100)
        - In SQLite these all map to the type TEXT
            - No size limit even if you specify one with VARCHAR
    - Decimal
        - Decimal types usually specify a precision and scale
            - DECIMAL(5, 2)
            - NUMERIC(10, 5)
        - Precision: total number of decimal numbers to store
        - Scale: number of decimal numbers to the right of the decimal point
        - Note that database can have very high precision decimal numbers

        - May be “lossy” when converting to things like a double
    - SQLite has two decimal related types:
        - REAL - double precision floating point number
        - NUMERIC - I believe double precision as well, hard to tell
- Dates
    - Typically multiple Date types
        - DATE
        - DATETIME
        - TIMESTAMP
    - Careful with timezones!
    - Typically stored in GMT
    - SQLite stores these as NUMERIC
- Booleans
    - Usually
        - BOOLEAN
    - SQLite, again, stores these as NUMERIC
        - KISS
- Blobs
    - Used to store raw binary data
        - BLOB
    - SQLite actually has a BLOB data type
    - A common alternative:
        - Write to disk or the cloud
        - Store URL for blob
        - Databases usually aren’t very good with binary data
- Column constraints
    - Additionally constraints may be placed on columns
        - PRIMARY KEY
        - NOT NULL
        - UNIQUE
    - We will discuss more in the next lecture
- Altering tables
    - ALTER TABLE statement
    - Syntax:
        - ALTER TABLE <name>
            - <alterations>
    - Here is a table rename
    - Add Col
        - To add a column you can use the ADD COLUMN clause
        - Common operation as your application grows
        - Some web frameworks manage these sorts of changes with a migration management system
    - Rename Col
        - To rename a column you use the RENAME COLUMN clause

        - Uncommon in my experience
    - Drop Col
        - Some databases support a DROP COLUMN clause
        - SQLite does not
        - You will need to
            - Create a copy table
            - Move data to the copy
            - Rename tables
            - The ol' LeadDyno gambit
    - Dropping Tables
        - SQLite does support dropping tables
        - Careful!
        - Remember to sanitize all user input!
    - Views
        - A view is a result set of a stored query
        - A view allows you to embed a query directly in the database
            - The DBMS may be able to optimize access to this data
            - Can encapsulate complex queries and joins to simplify data access
        - Views are read only
        - Creating
            - We use the CREATE VIEW statement to create views
            - Here we are doing some joins to display more friendly data when we look at tracks
        - Using
            - You can run queries against views
            - Here we find all tracks by AC/DC in this view
        - Deleting
            - Same as tables
        - Views
            - Let's them create the SQL just right…
        - Developers not so much
            - Inflexible
            - Difficult to update
- In class exercise
    - Let's go through an exercise with DDL
        - Design a simple schema via an ER diagram
        - Create DDL
        - Suggest modification
        - Create Update DDL
- DDL - Tables & Views
    - We looked at how to create tables with the CREATE TABLE statement
    - We looked at the common database column types
        - SQLite has a simplified set of data types

        - Commercial databases have much more extensive data types available
    - We looked at how to alter and delete tables and columns
    - Finally, we took a look at views
        - A way to encapsulate queries in a database so that they look like tables

DDL - Data Definition Language: Constraints on Data

- Last lecture
    - DDL: Data Definition Language
        - A language that defines your relational schema
    - We discussed how to define tables and columns with specific data types
- Constraints
    - A column type is a constraint on the column
        - Much the same way that a statically typed language like Java places constraints on the types of variables
        - Recall that SQLite is pretty loosey goosey with types
    - Most DBMS allow you to place additional constraints on columns
- Primary Key
    - Recall that a primary key is a column or group of columns used to uniquely identify a row within a table
    - By adding the primary key constraint to a column you are declaring that that column is the primary key for the table
        - It has a unique value for each row in the table
    - Multiple Cols
        - If you wish to declare multiple columns to be the main primary key for a table, you can use the PRIMARY KEY(...) syntax
        - Rare in practice, given the prevalence of synthetic keys
    - Nulls
        - The SQLite specification states that primary keys must not be null
        - For historical reasons, SQLite does not enforce this
        - You must also add a NOT NULL constraint to the key column to get the specification behavior
        - Note that the NOT NULL constraint is general and can be applied to any column
        - Here it is applied to Title as well
    - SQLite
        - SQLite automatically creates a rowid column for tables
            - Automatically get a new unique ID for each row inserted
        - If you have a single PRIMARY KEY column of type Integer, it will become an alias for the rowid column
- Foreign Key
    - Recall that a Foreign Key is a key in a table that refers to the primary key in another table
    - In this table, we have an ArtistId column that refers to ArtistId in the artists table
    - Constraints

        - Foreign Key (FK) constraints are important for maintaining referential integrity
        - To declare a FK constraint, we use the FOREIGN KEY declaration in the table definition
            - NB: column definition is optional, PK is the default
        - Important Note: SQLite, again, for backwards compatibility, does NOT enforce FKs by default
        - To do so, you must emit a PRAGMA statement
            - PRAGMA is a “meta” directive to the database
        - UPDATE and DELETE behavior
            - You can tell a database how to behave when a value referred to by a FK is updated
            - From a practical standpoint it is extremely rare to update a value referred to by a FK
            - However, deleting rows referred to by an FK is common
        - CASCADE - when a row referred to by a FK is deleted, delete all rows in this table that referred to it
            - A delete in the artist table cascades to the albums table
            - To the tracks table as well?
                - No you need to declare it on all FKs
            - Can make deletes expensive
                - Another reason to prefer archiving
        - NO ACTION & RESTRICT - do not allow a delete from the referred to table if any row has an FK pointing to it
        - You must programmatically remove all data in the appropriate order
            - I prefer this approach
        - SET NULL - on a delete on the referred to table, set the value of the FK to null for all rows that refer to that row
            - Might be useful for something like genre, where if you remove a genre then Tracks revert to null as “unknown”
        - SET DEFAULT - on a delete on the referred to table, set the value of the FK to the default value specified for that row
- Unique Constraints
    - You might use UNIQUE constraints to enforce sensible data in non-PK columns
        - Good example: email
    - Shows a weakness of the synthetic key model:
        - It’s possible to get nonsense data if you don’t use additional constraints
- General Constraints
    - SQLite allows you to define general logical checks on your data using the CHECK() clause
    - The expression cannot contain a sub-query
        - This makes it of limited use in most practical cases

        - Auto Increment
            - You can declare a column to AUTOINCREMENT
            - The algorithm SQLite uses in this case is slightly different than rowid
                - It does not reuse IDs
                - More CPU intensive
            - Generally, SQLite recommends against using this feature
    - DDL - Constraints
        - Today we took a look at various constraints you can place on columns in your DDL
        - The most important, by far, is the ForeignKey constraint (FK)
            - Delete behavior matters!
        - There are also ways to constrain nulls, uniqueness, etc…
        - In most systems, data integrity is maintained at both the DB and the application level
            - You are not going to be verifying email formats, for example, in the DB

Transactions: Ensuring Data Consistency

- Transactions in SQL
    - Thus far we have treated operations as if each was independent
        - The operations are executed one at a time
        - Each one is “atomic” in that it succeeds or fails entirely
- ACID Properties
    - ACID properties of databases
        - A = Atomicity
        - C = Consistency
        - I = Isolation
        - D = Durability
    - Transactions help in ensuring these properties
- Atomicity
    - Atomicity: an operation either completes entirely or fails entirely
        - If the DBMS goes down half way through the Track Insert, when the DBMS resumes it will be as if the insert never happened
    - Atomicity can also apply to multiple statements
    - Here we are transferring 10 bytes from one track to another
        - Just pretend, OK!
    - What happens if the DBMS goes down after the first operation but before the second?
    - What if this happened with bank accounts?
- Consistency
    - A logical constraint on this operation is that the number of bytes in the tracks remains the same
        - By failing atomically we also fail from a consistency standpoint
- Independence
    - Consider two users running this update at the same time

        - Both operations must complete as if the other operation was not concurrently running
        - E.g. user 2 cannot accidentally read the bytes value before user 1 and then write after user 1
            - We will discuss serializability in a bit
- Durability
    - If the system crashes after it has returned success for this transaction, when it restarts it must show the results of this transaction
        - This is often implemented with what is a log file
- ACID Summary
    - ACID is a property that is achievable with databases but not guaranteed
        - ACID is expensive to guarantee
    - You typically use transactions to achieve some set of these properties
- Non-ACID Idea: Serializability
    - Operations on your data must take place in a logically serializable order
    - Consider two people trying to reserve the same seat on a plane
        - User 1 sees the open seat
        - User 2 sees the open seat
        - User 2 reserves the open seat
        - User 1 reserves the open seat
- Serializability
    - Serializability: constraining this sequence of events to act as if each see/reserve action took place serially, rather than in parallel
    - Many techniques for achieving serializability
        - Locking: the most common
    - Related to independence
- Transactions
    - Again, the DBMS solution to provide all of these properties are Transactions
    - A transaction is a collection of operations that must be executed atomically
        - All succeed or all fail
    - Transaction syntax:
        - BEGIN TRANSACTION or START TRANSACTION to start a transaction
        - COMMIT to commit a transaction to the database
        - ROLLBACK to abort a transaction
    - By default, most DBMS are in Auto-Commit mode: when a SQL statement is not enclosed in a pair of start-transaction (BEGIN or SAVEPOINT) and end-transaction (COMMIT, ROLLBACK, or RELEASE) SQL statements, it is executed in the database transaction implicitly delimited by the boundaries of the SQL statement. The SQL statement is said to be in auto-commit mode, since its database transaction is automatically delimited.
    - SQLite operates in auto-commit mode when you interact with it in JDBC
- Isolation levels
    - Transactions can have different isolation levels

        - Isolation levels determine how a transaction interacts with other concurrent transactions
        - Allow you to pick what level of ACID you wish for your app
            - Trading off against performance
        - SERIALIZABLE
            - Most restrictive isolation level
            - All reads and writes are locked
            - Additionally, and ranges scanned in a WHERE clause must be locked
                - This prevents Phantom Reads where data is missed because it is being inserted concurrently
            - Can be very expensive
        - REPEATABLE READS
            - All reads and writes are locked
            - No range locking
                - Phantom Reads can occur
            - Less expensive than SERIALIZABLE
            - Called repeatable reads because the data could be re-read at any point in the same transaction and give the same results
        - READ COMMITTED
            - Writes are locked
            - Reads are only guaranteed to be part of a committed transaction
            - No range locking
                - Phantom reads can occur
            - Less expensive than REPEATABLE READS
        - READ UNCOMMITTED
            - Writes are locked
            - Reads may be part of an uncommitted transaction
                - So called Dirty Reads
            - No range locking
                - Phantom Reads can occur
            - Least expensive, also the fewest guarantees
        - Setting the transaction isolation level is different per database, but typically looks something like this
            - SET TRANSACTION
            - ISOLATION LEVEL
            - SERIALIZABLE
- Isolation levels in SQLite
    - Transactions in SQLite are SERIALIZABLE
        - SQLite supports multiple simultaneous read transactions coming from separate database connections, possibly in separate threads or processes, but only one simultaneous write transaction
    - This is not good for throughput in high-write applications
    - Other databases are going to perform much better for high-write workloads
- Locking for transactions

        - To implement transactions, databases must implement specific locking strategies
        - SERIALIZABLE must acquire read locks on all data (and ranges) read, as well as write locks on all updated data
- Lock Types
    - Read locks can typically be held by multiple threads
        - We are all reading this data, and that's ok
    - Write locks can only be held by one thread
        - Cannot be acquired if a read lock is already present on the data
- Lock Issues: Contention
    - Lock contention occurs when many database sessions all require frequent access to the same lock
        - Typically a write lock scenario
    - If you have a "hot row" in your database that is updated frequently it can hurt throughput
- Lock Issues: Blocking
    - Lock blocking occurs when a transaction holds a lock for a long period of time
    - A long running batch processes may, for example, cause your web site to lock up waiting for a read lock
- Lock Issues: Deadlock
    - Deadlocks occur when two processes hold locks, and each needs the lock that the other holds
    - Deadlocks are detected by some DBMS these days and one or the other transaction is aborted
    - To solve deadlocks, you can acquire locks in a specific order
        - E.g. acquire locks by table name
- Transactions
    - Transactions are used to guarantee data consistency
    - Depending on your consistency needs and DBMS, there are varying levels of isolation available in transactions
    - Transactions use locks to ensure data consistency
        - Issues arise from locking
            - Contention
            - Blocking
            - Deadlock

Optimistic Concurrency: the real world

- Transactions
    - In the last lecture we discussed transactions as a way to ensure data consistency
    - As a motivating example, we considered transferring money between accounts
    - Today let's consider a similar problem, booking seats on an airplane flight
    - Here we have two users, each considering the seats that they want
- An online transaction
    - SERIALIZED logic:
        - Person A reads (views) the available seats
        - Person B reads (views) the available seats

        - Person A attempts to book seats
            - FAIL: Person B has a read lock
    - What if Person B….
        - Leaves the browser window open in a tab and forgets about it?
        - Closes the window?
        - Refresh the window?
    - READ UNCOMMITTED logic:
        - Person A reads (views) the available seats
        - Person A writes (reserves) a seat
        - Person B reads (views the available seats) before COMMIT
        - Person B sees seats as available that are not actually available
    - The reality here is that we have a transaction that is spanning multiple web requests
    - Transactions are expensive and, with the web, we don't know if users will ever complete their action!
- Pessimistic Concurrency
    - Transactions are a form of pessimistic concurrency
    - They assume things will go badly and use a locking/serialization strategy to ensure that they don't
    - Pessimistic concurrency
        - Complex implementation
        - Locks
        - Deadlocks
        - Etc…
    - Is pessimistic concurrency necessary?
    - Is pessimistic concurrency web friendly?
    - Database people tend to be pessimistic
- Optimistic Concurrency
    - There is another option available, however: Optimistic Concurrency
    - Allow concurrency issues to happen, but…
        - React to them when they do
        - Assume that, for the most part, things will work out
    - Developers tend to be optimists (aka "naive")
    - At a high level, optimistic concurrency can be thought of in the following way
        - Don't lock reads!
            - Allow people to read data freely without acquiring any sort of read lock
        - When a write occurs, "check your work" to make sure no other changes have occurred
    - Implementing O/C
        - There are many ways to implement optimistic concurrency
        - A simple (to understand) implementation would be:
            - When issuing an update, include all the current values of columns in the WHERE clause

- Here we are updating the artist “AC/DC” to have the name “DC/AC” instead
- Rather than just using the ArtistId column in the where clause, we use all values in the where clause
- Now, after the update executes, we can check the number of rows updated
- If 1 row was updated, we have succeeded: no one has updated AC/DC in the meantime
- No more than one row should be updated… why?
- In that case, our optimism has paid off: we’ve managed to update the artist row without any complex transaction logic
- What about the failure case, where 0 rows are updated, what does that mean?
- Oh well, that artist has already been updated
- Maybe we can just let the user know in the UI
- Refresh the data… maybe they will try again
    - Have you ever had this happen on a website?
- This approach to concurrency is very web friendly
- If a user is looking at tickets and wanders away or their internet goes down or whatever, no read locks are being held
- Other people can still make a claim on the resource
- If there is a concurrency issue and a user accidentally picks a seat already taken, is the situation worse than with stricter concurrency?
- Not really, one and only one user will win, which is the same outcome
- This simple version of optimistic concurrency works but it’s a little expensive: have to keep all the data for a row around to update it
- Another approach would be to use a version column in your schema
- Here we bump the version on every update
- Has the advantage that we don’t need to send every column value in the WHERE clause, less chatty with the database
- Also a little more conceptually clear
- Using a version column is a very common approach and is supported in many database frameworks
- Java standardized this in the JPA - the Java Persistence API
- The JPA is is an extension to JDBC
- JDBC: low level, raw SQL access
- JPA: high level, object relational mapping API
- In our project we are working with JDBC, the lower-level of the two APIs
    - Better experience for you in my opinion
- JDBC code for optimistic locking
- Prepare an update with all fields in the WHERE clause
- Call executeUpdate()
- Check to see if one and only one row updated
- Spring data is an object relational mapping system

        - We will discuss O/R mapping in more detail in a later lecture
    - Here is an example of an optimistically locked data object
    - The @Version annotation tells spring to use the version field for optimistic locking
    - On a failure, Spring will throw OptimisticLockingException
    - It is up to the application developer to anticipate these exceptions and handle them properly
    - Other potential optimistic concurrency implementations I have seen or heard of:
        - A timestamp column (updated at)
        - A random number
        - A cryptographic hash of the entire row
    - Note that while optimistic concurrency removes the need for locking between application requests, we still need transactions within a request
        - We can't get around the need to make some operations atomic
    - This can be difficult for people to understand: "why do I need pessimistic locking if I have optimistic locking?"
    - What if we were swapping two teams names?
        - Unrealistic, but bear with me!
    - What if the first update succeeds with one updated row but the second one fails?
    - As it currently stands, the first update will be in the database, but the second one will not
    - Two teams with team2Name!
    - To fix this, we need to wrap the entire optimistic concurrency implementation in a transaction
    - Turn off autocommit
    - Roll back the transaction if any of the updates fail
    - Otherwise, commit
    - Very common for this pattern to be wrapped up in an API of some sort:
        - Gather all updates into a single "bundle"
        - On "commit" of the bundle, start transaction
        - Iterate through bundle issuing UPDATEs
        - Roll back if any fail
        - Commit if all succeed
    - You will be implementing optimistic concurrency for one class in the Project for the Artist table
    - You will not need to deal with Transactions, because there are not multiple updates in play
- Academics
    - It turns out optimistic concurrency is not discussed much in the database literature
        - Probably due to its "ugly" and pragmatic nature
    - Despite this fact it is used everywhere in the real world

        - A good example of the real world/academic distinction
- A third option
    - There is a third concurrency option: Valhallocking*
        - Just ignore the problem
        - I live, I die, I live again
    - A surprising number of web applications have taken this approach
        - Advantages: very fast
        - Disadvantages?
- The real world
    - In the real world, most online systems are either optimistically or valhallocked
        - Shoot for optimistic locking when concurrency is a real issue
        - Don't feel bad about valhallocking if concurrency isn't an issue
- Optimistic Concurrency
    - Transactions are useful for keeping data consistent, but have issues with long-lived and uncertain workflows
    - Optimistic concurrency fits better with the web (and mobile) model with disconnected clients
        - Assumes things will probably work out
        - Column values are used to detect a bad update
        - Transactions are still used within a given request, but not across requests
    - Valhallocking isn't the end of the world for MVPs and many smaller applications

Object Relational Mapping: Tools for working with Databases

- Object Relational Mapping
    - The object/relational mapping problem is the problem of mapping in memory objects to relations stored in a DBMS
    - A system that does this management is called an Object Relational Mapper (ORM)
    - Thus far in our projects, we have been building an ORM ourselves, by hand
        - This is done so that you can see how things are working at the SQL level
    - Professionally, you will most likely be working with an ORM of some sort
    - Top: raw SQL accessing relational data directly from a database API
    - Below: the same logic, using an ORM
    - Looks a lot nicer, doesn't it?
    - And it is…
        - Typically less code
        - Often has better code-tooling support (e.g. autoComplete)
    - However
        - You are less connected to the underlying database
        - Easy to cause performance issues without realizing it
    - In the Java world, there are several options for ORM frameworks
    - The oldest one that I'm aware of is Hibernate
        - First released in 2001
        - Developed by Cirrus Technologies, then JBoss, now Red Hat

- I don’t particularly care for Hibernate, but it is a standard and demonstrates most things an ORM will do
- Other alternatives
    - Spring
    - jOOQ
    - Apache Cayenne
- Hibernate: Architecture
    - Hibernate Architectural Model
        - Data Access Layer: Java objects
        - JPI/HNI: Database abstraction layer
        - Hibernate: core Hibernate
        - JDBC: the Java Database Connectivity API
            - This is what we are using
        - The relational database
- Hibernate: Mapping
    - Consider this contact table
        - ID field (Primary Key)
        - First, last, middle name
        - Notes
        - Starred
        - Website
    - Hibernate uses java annotations to provide metadata
        - This helps hibernate map the fields in the Java object to the database
        - Very common approach for metadata in Java frameworks
    - Annotations
        - Java Annotations are a way to provide metadata about a feature (methods, fields, etc…)
        - Can be used reflectively at runtime to ask if a feature has a given annotation
        - DEMO
    - @Entity - this class is an entity that maps to the Contact table
    - @Id - this the primary key for this table
    - @GeneratedValue - this value is automatically generated by the database
        - Shown in a few slides
    - Note that the Name class has been pulled out and annotated as @Embeddable
        - The object oriented concept does not match the database implementation
        - Columns are directly in the contact table, but are modeled in a more OO approach here
        - More on this next lecture
- Hibernate: Associations

        - Associations describe how two or more entities form a relationship by providing join semantics for the relationship
        - Here we have a phone, which has a Many-to-one relationship with the Person class using the @ManyToOne annotation
        - Note the use of the @JoinColumn annotation to describe the foreign key to be used in the relationship
        - List associations are defined using the @OneToMany annotation
            - Hibernate supports join tables or direct references
            - Here it is a direct reference
                - Note that it will programmatically cascade deletes
                - And will also remove any “orphans” - any Phones that have a null person
        - Note that this association does not specify any join attributes
        - Rather is defers to the person property on the phone, on the other side of the 1-to-many relationship
            - Subtle, and a little annoying
        - Given this java code…
        - This SQL will be executed
        - What’s the difference between flush() and persist()?
        - persist() - hibernate, here is some data for you to save
        - flush() - hibernate, make sure that all the changes you have, has been synchronized with the database
- Hibernate: Transactions
    - Hibernate supports transactions
        - The API is pretty terrible, I omitted a bunch of code…
        - The begin() and commit() methods start and commit the transaction
    - Side rant: this is the problem with the Java community
        - API designers just can’t get out of their own way and build an API without a ton of builders, factories, and so forth
        - A legacy of the J2EE era
        - Too bad, java is a pretty good language and the JVM is awesome
- Hibernate: Querying
    - Hibernate offers a bunch of different ways to query data
    - We will focus on two
        - Native SQL
        - HQL
    - Native querying uses the native query syntax of the backing database
    - Produces a list of the type given as an argument
- Hibernate: Parameters
    - Adding parameters is simple as well
    - Note that parameter is name-based rather than index based, as in raw JDBC
- Hibernate: Eager Loading

        - A person has multiple phones
        - We may wish to avoid multiple queries to display this information
        - Hibernate can eagerly load collections
            - One query for all the info
- Hibernate: HQL
    - Hibernate has implemented its own query language, HQL
        - Similar to SQL
        - Object oriented
    - Supports niceties such as
        - Omitting some unnecessary syntax
        - Allows you to use polymorphic information in your queries
- Hibernate: Caching
    - Hibernate has multi-level cache infrastructure
    - First-level cache
        - Multiple updates to an object will be kept until an update (flush()) occurs
    - Second-level cache
        - An optional, pluggable cache layer
    - Query-level cache
        - An optional layer that caches query results
    - Caching can be an extremely important aspect of system performance
    - But remember:
        - “There are only two hard things in computer science: cache invalidation and naming things”
            - Phil Karlton
- Optimistic Concurrency
    - “The only approach that is consistent with high concurrency and high scalability, is optimistic concurrency control with versioning”
        - Hibernate Docs
    - Note the use of the @Version annotation
    - This alerts Hibernate to use optimistic concurrency in updates
    - Hibernate will now emit SQL that looks like this when updating Order objects
    - You may have noticed slightly different annotations here
    - This is using the JPA annotations rather than the native Hibernate annotations
        - Welcome to Java!
- Yet another O/R library
    - You will be surprised to learn that I do not like the way that Hibernate (or most other ORMs) work
    - I have, therefore, started working on my own version
    - ChillRecord
        - A play on ActiveRecord, the Ruby on Rails O/R system, which I like

        - Uses code generation instead of annotations
        - Still very early in life, but interesting nonetheless
        - We'll look at it in-depth in the next lecture
- Object Relational Mapping
    - Today we discussed what ORM systems are
        - A tool for managing objects in memory and mapping them down to relations in a database
    - We took a look at Hibernate, a popular OR framework for Java
        - Defining entities
        - Working with entities in code
        - Querying entities
    - Next time, we will discuss problems with ORMs
        - The Object-Relational Impedance Mismatch
- Test Relevance
    - In the project we are not using an ORM
    - In fact, we are building a rudimentary ORM
    - I do not expect you to know anything about Hibernate in particular for the final
        - I am showing you this to prepare you for a future job, the O/R tool you will end up using will most likely not be Hibernate
    - I do expect you to understand what an ORM is
    - Next lecture, I will discuss many of the problems with ORMs and I will expect you to understand those problems

The O/R Impedance Mismatch: Why some folks hate ORMs

- Object Relational Mapping
    - As we discussed last time, the Object/Relational Mapping problem is the problem of mapping in memory objects to relations stored in a DBMS
    - We took a look at Hibernate, a mature OR framework for Java
    - Seems like a pretty good tool, right?
    - Better than writing SQL in string literals, anyway?
    - Not everyone agrees…
- ORM Hate
    - "While I was at the QCon Conference in London a couple of months ago, it seemed that every talk included some snarky remarks about Object/Relational Mapping tools (ORM)" - Martin Fowler
    - "Object/Relational Mapping is the Vietnam of Computer Science" - Ted Neward
    - "All the solutions they come up with seem to make the problem worse. I agree with Ted completely; there is no good solution to the Object/Relational Mapping problem" - Jeff Atwood
- The O/R Impedance Problem
    - Why all the complaining?
    - Because of a class of problems that has come to be known as the OBJECT/RELATIONAL IMPEDANCE MISMATCH
    - Objects relate to one another in a GRAPH

        - Object A has a pointer to Object B, and navigates it via memory references
            - This is true even in Java
        - Relational schemas are tabular
            - Based on relational algebra
- OO Concepts
    - Encapsulation
        - Internal representation of data is hidden
        - Mismatch: compare with a database row: all the data is “public”
    - Accessibility
        - OO may have a well developed sense of accessibility
            - Public
            - Private
            - Protected
        - Mismatch: compare with a database row: the data is there, or it isn’t
    - Polymorphism
        - OO has a notion of polymorphism: objects may share a base class or implement a common interface
        - Mismatch: no native notion of interfaces or inheritance in the relational model
        - Recall that we discussed three mechanisms for dealing with polymorphism at the DB level
            - Single-table inheritance - One table for all subclasses
            - Table per concrete class - one table for each concrete subclass
            - Table per class - one table for each class
        - Each mechanism had tradeoffs that had to be accepted
        - No one canonical, strictly better solution to the problem
- Data type differences
    - Database and programming languages often have different notions of both types as well as type rules
        - Fixed precision numbers in a DB do not map cleanly to floating point
        - Strings may behave differently with respect to white space
        - Etc..
- Integrity differences
    - Database and programming languages often have different notions of integrity
        - An object may allow a reference to be null for temporary reasons
        - Whereas a table may not allow null on the related relation
- Conceptual differences
    - Databases work in SQL, which is a fundamentally declarative language
        - Tell the database what you want, not how to do it
    - Most programming languages that work with databases are imperative
        - Tell the program what to do, step by step
- Transactional differences
    - In databases, transactions, the smallest amount of logical work can be very large

        - Compare with OO programming, where a transaction is typically a single operation
            - OO languages typically have no notion of isolation or durability
- Philosophical differences
    - Declarative vs Imperative
    - Nouns & Verbs
        - OO encourages a tight binding between data and actions
    - Identity
        - Primary key vs. Object identity
    - Normalization
        - Not a feature of OO
    - Set vs. Graph
- Equality
    - What does equality mean in an OO system?
    - What does it mean in SQL?
    - If two objects have the same ID, are they equal?
        - This is a generally hard problem in OO
- The O/R Impedance Mismatch
    - All of these problems, taken together, form the O/R Impedance Mismatch problem
    - How much of a problem is it?
    - “ORMs help us deal with a very real problem for most enterprise applications. It’s true they are often misused, and sometimes the underlying problem can be avoided. They aren’t pretty tools, but then the problem they tackle isn’t exactly cuddly either. I think they deserve a little more respect and a lot more understanding.” - Martin Fowler
    - “I’ve come to the conclusion that, for me, ORMs are more detrimental than beneficial. In short, they can be used to nicely augment working with SQL in a program, but they should not replace it.” - Geoff Wozniak
- Dealing with it - Solutions
    - Many proposed solutions
        - “Better” ORMs
        - “Simpler” ORMs
        - NoSQL
        - Object-oriented database management systems
    - Jooq
        - “Database First”
        - Focuses on generating SQL
            - Not abstracting away the database
        - Generates code that represents the relational schema in Java terms
        - Allows for a more straightforward mapping to the underlying DB
        - Not nearly the life-cycle functionality of other ORM systems
        - “Closer to the metal”
            - Good?

        - Bad?
- LINQ (Microsoft)
    - Microsoft introduced LINQ (Language Integrated Query) to .NET in 2007
    - Integrated Query language directly into the languages
    - Can work with many different providers
        - LINQ to SQL works with DBMS
        - Can also work with XML documents or in memory objects
    - At right we can see that the query is integrated directly into the language
        - In a Microsoft IDE, you get code completion, find usages, etc…
    - Additionally this sort of syntax can be used against an array, or an XML document, etc…
    - Here is how you insert a user object into a database
    - Note that there isn't any special syntax here compared with other O/R solutions
    - The embedded syntax is nice
        - Really nice, actually
    - But it only really solves the query problem
        - And only part of the query problem
        - What if you want to conditionally include some criteria?
            - E.g. our advanced search dialog
    - Ultimately, although it is a great technical achievement and interesting programming language problem, I don't think LINQ solves the O/R impedance mismatch
- Active Record
    - ActiveRecord
        - Originally out of the rails community
        - Old, very mature
        - My favorite ORM
    - Allows validations to be declared in model classes
        - Many useful validations
            - Regexp
            - Presence
            - Etc…
    - Life cycle callbacks allow you to insert logic over the life cycle of the object
        - E.g. after insert, denormalize some data to another table
    - Query syntax is nice
        - 90% solution for most application queries you have
        - You can bail out to SQL if necessary really easily
    - Migrations
        - Has a baked in notion of schema migrations
        - As your DB is updated, you write migrations from one state to the next
        - Your DBs can be migrated over time

                - Dev
                - Staging
                - Production
            - Other niceties
                - You can get the old values for columns before you save
                - Good, explicit caching support
                - Doesn't try to do too much, allows you to bail out to SQL when necessary
            - My take: ActiveRecord is the best ORM solution I have worked with
                - Not perfect, but good enough
                - Gets out of the way where necessary
                - Provides a lot of value outside of the O/R mapping problem
        - ChillRecord
            - My ActiveRecord-inspired O/R tool
            - Not yet released, barely works
            - May eventually integrate this into 440, we'll see!
            - DEMO
- The O/R Impedance Mismatch
    - The O/R Impedance Mismatch: problems caused by trying to integrate the Object and Relational models with one another
    - Different world views
        - Relations vs. Objects
        - Declarative vs. Imperative
        - Tuples vs. Polymorphism
        - Etc…
    - Some possible solutions to the O/R problem were examined
        - Jooq
        - Linq
        - ActiveRecord
        - ChillRecord

DDL - Indexes and More: Indexes, Triggers, and Stored Procedures:

- Previously we discussed how to express constraints on data columns in a database
- Things like
    - Declaring a column a primary key
    - Declaring a column non-null
    - Declaring a column a foreign key
    - Etc…
- Indexes
    - Recall that in a database, a table is a set of rows
    - Rows are laid out sequentially on disc
    - In SQLite, unless you say otherwise, all rows have an associated rowid
        - An integer declared to be primary key will be an alias for this column
    - Recall the EXPLAIN command
    - Tells us how a given query is going to execute

    - The database often does a good job of figuring out the best way to execute a query
    - But sometimes it needs some help
    - We can hint to the database that a certain query is going to be run consistently
    - We can add something called an index to help the database execute those queries more efficiently
    - An index is just a secondary data structure that helps improve the performance of queries
    - It lives alongside the database table and provides information that helps the query engine make smarter decisions
    - SQLite uses B-trees for maintaining indexes
        - B-Tree stands for Balanced Tree (not Binary Tree)
        - B-Trees allow for efficient (O(log n)) queries using relational operators
    - We will discuss B-Trees in more detail in the next lecture
    - At right is a graphic example of a B-tree
        - Side note: this B-tree has pointers between leaf notes, making it a B+ tree
        - Efficient find as well as ordered iteration
    - Indexes are associated with a specific table
    - Indexes are added on specific columns in the table
    - A big part of development with a database is figuring out where you need indexes
    - Serious performance boosts are possible!
    - Adding an index uses the CREATE INDEX statement:
        - CREATE [UNIQUE] INDEX
        - <index_name> ON
        - <table_name>(<column_list>)
    - Note that indexes can place UNIQUE constraints on columns
    - In a sense, indexes are a specialized constraint put on columns
        - Useful particularly in multi-column indexes discussed in a bit
    - Here we are adding an index on the email column in the employees table
    - EXPLAIN QUERY PLAN before and after demo…
    - Note: an index on a text column like this is useful only on:
        - Relational queries
        - Prefix searches
    - General text search will still cause a table scan
        - Why?
- Indexes - multicolumn
    - Consider a query against the playlist_track join table and an associated index
    - Is this index helpful?
    - Yes, it still narrows down the set of nodes that need to be looked at
    - Not as good as a multi-column index though!
    - Here is a multi-column index
    - This index is perfectly tuned for this query
        - O(log(n)) search time
        - No additional scanning

    - SQLite is smart enough to add this index automatically because both fields are part of the primary key
        - Look for indexes named sqlite_autoindex
- Indexes - foreign keys
    - Foreign keys almost always need an index to perform well
    - Note SQLite does not automatically add FK indexes, so they must be added manually
    - So, for example, a common query in your web application might be “find all tracks on a given album”
    - Without adding an index, this will involve a table scan every time you make this query
        - With some caching by the DB
    - To alleviate that, you will need to add an index like at right
        - Indexing the tracks table on the AlbumId field
    - Now searching for all the tracks on a given album has gone from O(N) where N is the number of tracks to O(log(N))
    - A big deal for large N!
    - In our application you will hardly notice it, N isn’t big enough :)
- Indexes - Performance
    - Much of performance work in online applications is database tuning
    - Adding and tweaking indexes
    - The indexes increase read performance
    - However, there is a tradeoff
    - Can anyone guess it?
    - Indexes also can hurt insert, update and delete performance
    - DB must now do more work on those operations:
        - Mutate the table
        - Also mutate the associated indexes
        - Must be done automatically!!
    - So you don’t want to just go index crazy
    - Especially if your application workload is write heavy
    - Judicious use of indexes are extremely important and a bit of an art
        - Profiling helps here!
- Indexes - Expressions
    - SQLite supports expression indexes
    - Not a standard part of SQL as far as I’m aware
        - May prove useful to you at some point
        - Other DBs may have something similar
    - If you are using a computed value in an expression, indexing on the result of that expression may help with specialized queries
        - Relational or sorting operations
    - Another option here is to denormalize the column
    - DBAs may hate it, but it works…
- Triggers

- Triggers are a named bit of logic associated with a table and triggered on a certain event
- Syntax
    - CREATE TRIGGER [IF NOT EXISTS]
    - <trigger_name>
    - [BEFORE|AFTER|INSTEAD OF]
    - [INSERT|UPDATE|DELETE]
    - ON <table_name>
    - [WHEN condition]
    - BEGIN
    - Statements;
    - END;
- Here we are verifying the formal of the employees email in the database
    - Benefits to this verification approach?
    - Drawbacks?
- Denormalizing data example
- Addresses the same issue that we used an expression index for previously
- Trigger events
    - Table related
        - BEFORE INSERT
        - AFTER INSERT
        - BEFORE UPDATE
        - AFTER UPDATE
        - BEFORE DELETE
        - AFTER DELETE
    - View only
        - INSTEAD OF INSERT
        - INSTEAD OF DELETE
        - INSTEAD OF UPDATE
    - New and old symbols
        - INSERT
            - New is available
        - UPDATE
            - Both new and old are available
        - DELETE
            - Old is available
    - Advantages of triggers
        - Typically fast
        - Defined with your data model
        - Can’t be avoided via a side-channel
    - Disadvantages of triggers
        - Tend to be opaque
        - Difficult to track down from a non-DB environment
- Stored Procedures

        - Many databases have a feature called Stored Procedures
        - Stored Procedures are functions defined within a database
            - Can be very fast
            - Can take advantage of native database features
            - Database may heavily optimize stored procedures
        - SQLite does not support stored procedures
        - I have not seen heavy use of stored procedures in industry
            - Tends to be crucial logic in the database, away from the application
    - DDL - Indexes and Triggers (and Stored Procedures)
        - Indexes can be used to improve read performance
            - Key part of application performance tuning
        - Indexes tend to hurt insert, update, and delete speeds
            - As always: tradeoffs need to be made with empirical, realistic data
            - What is your application's data access profile?
        - Triggers can be used to execute logic after events in the database
            - Loved by DBAs, hated by developers
        - SQLite does not have stored procedures
            - A way of defining functions directly in the DB
            - Again, loved by DBAs, hated by developers

B Trees: Database Implementation

- In the last lecture, we looked at a crucial aspect of performance in databases: indexes
- In this lecture we will look at the primary data structure used for implementation of databases: the B tree
- B Tree History
    - Invented by Rudolf Bayer and Edward McCreight at Boeing
    - Rudolf Bayer
        - German professor
        - Also invented the UB-Tree and the red-black tree algorithms
    - Edward McCreight
        - US Computer Scientist
        - Also co-designed the Xerox Alto
- Why "B" Tree?
    - What does the B stand for?
    - Binary? No, B tree nodes can have more than two children…
    - Some other suggestions
        - Boeing
        - Balanced
        - Broad
        - Bushy
        - Bayer
    - Basically, we don't know, makes you learn more about B trees the more you think about what it means
- What is a B-Tree?
    - Well, first off, it is obviously a Tree

        - A collection of hierarchically arranged data
    - It is also self balancing
        - No branch can get too deep compared with other branches
    - As a tree, it supports logarithmic time operations
        - Search, insert, delete
    - B-trees are particularly well suited for block storage
    - Block storage is any storage system that consists of “large” chunks, or blocks, of data
    - The file system is extremely slow when compared with memory
        - Which is extremely slow when compared with L3 caches
        - Which is extremely slow when compared with registers
    - Because of this, when you ask for data from a spot on disk, you typically get a chunk
    - Latency of the request dominates the throughput cost of transferring additional data
        - Bias towards transferring more data
        - Hope that “locality” means that the data is useful
    - Same concept is applied in memory caches, etc…
    - A B-Tree node can be sized to the size of a disk page and, therefore, can be efficiently traversed
    - Because of this, the B-tree data structure is very common to see in disk related technologies
        - E.g. file systems
    - Pictured at right is a simple B tree
    - The B Tree consists of Nodes
    - Nodes hold Keys (values) and Pointers
    - The top node is the Root Node
    - The bottom nodes are all Leaf Nodes
    - Note that a node in this B tree has room for four values (keys) and five pointers
    - The branching factor of this tree is 5
        - Note that the number of values available is always (Branching Factor - 1)
    - The branching factor is sometimes called the degree of the B tree
    - Because of this relationship, you will sometimes see specific B trees described as “N-M B trees”
        - N = number of keys
        - M = number of pointers
    - In our example we have a 4-5 B-Tree
- B Tree Operations - Search
    - Search in a B tree is similar to other tree search operations
    - Search for value 12
        - Begin in root
        - Scan numbers
        - 12 is < 16, so follow pointer
        - Scan leaf

        - Find value at second position
    - O(log(n)) where n = number of keys
- B Tree Operations - Insert
    - Start at root node
    - Find leaf where value is to be inserted
        - If node contains fewer than max values, insert it
        - Otherwise, split into two nodes
            - Push median value to parent
            - Less than median to right
            - Greater than median to left
    - Here we have a 2-3 B Tree
        - 2 keys
        - 3 pointers
    - Insert values 1-7
    - On each insert, we either add the key OR split
    - Note on last insert, we recurse on splitting, creating a new root node
- B Tree Operations - Deletion
    - Deletion is a more complex algorithm
    - The first concept to know is a fill factor
        - This is the fewest number of elements a node is allowed to hold
    - Here we have a 2-3 b-tree with a fill factor of 0
        - A node cannot have 0 elements
    - Case 1: deleting a leaf value when there is no fill factor violation
    - Consider deleting the number 32
    - This is easy: simply remove the value
    - Case 2: deleting a leaf value when there is a fill factor violation
    - Consider deleting the number 31
    - Now we have a violation: the node that held 31 has no values in it
    - We borrow a value from the left sibling of the node
    - Note the left sibling has the values 25 and 28
    - We don't want to just grab the value 28 and move it over, since it is less than 30
    - So we rotate the value 28 up to the parent and bring the parents value, 30, down
    - If the left sibling node does not have enough numbers to borrow from, we check the right sibling and do a right rotation
    - What if neither sibling has enough values to borrow from?
    - We merge the siblings through the parent node
    - If we delete 30 in this case, we merge 28 from the parent down with 25 from the left sibling
    - What about internal values?
    - Consider the easy case: the child node to the left of the deleted node has enough values to donate
    - Simply move the right most number from the left child up
        - Or vice-versa with the right child
    - If both children have the minimum values possible, merge the two children

        - Here, when deleting 30, the left and right child only have one value
        - We merge them together as a single node to the left of 32
        - Final case to consider: what if an element is deleted from a node that drops it below the load factor and no values are available to borrow
        - Here we have to shrink the tree
        - Consider deleting 10 from the tree to the right
        - We can't simply move 5 or 15 up
        - We also can't rotate the right sibling over
        - We must instead merge 20 and 35 into a new root node
        - The original children become the left-most child of the newly formed root node
        - Note that the tree shrunk in height
        - This is the self-balancing nature of B-trees
- B+ Tree
    - A B+ Tree is a variant of the B Tree
        - Includes all key values in leaf nodes
        - Contains a next pointer to point to the next node at a given level
        - Typically have very high degree (100+) when compared with B-Trees
    - Note that the next pointer makes it very fast to do an ordered iteration over all keys
        - No need to visit parent nodes, just head to the first leaf node and let it rip
    - Databases typically use B+ Trees
        - In fact, they are so common, people will say B tree when they mean B+ tree
- Database usage
    - What are B+ Trees used for in databases?
        - Pretty much everything!
        - Tables are often laid out in B+ trees
            - Rowid is the key
        - Indexes are smaller B+ trees
        - Views can also be implemented with B trees
    - Here is a table laid out on disk with a B+ tree based on the rowid entry
    - An index on a given value (e.g. Age) would look similar
    - Note that in reality, multiple rows in the DB would be in a single node
    - Consider this query
        - With no index, we must scan the entire table testing the condition
        - O(n) time to compute this
    - If we have an index (B+ tree) on milliseconds, how can it be implemented?
        - Using B+ tree, search for first leaf entry > 18000
        - From there, follow the B+ tree pointers to end
        - O(log(n)) time
        - Much faster
        - This is the crux of how database indexes work
- Why B-Trees?
    - Ok, so why use B-Trees instead of other potential data structures?

        - E.g. serialized hash tables or binary trees, etc…?
    - Well, we still need to scan or at least do a binary search of the individual nodes for many operations
        - Not just direct lookup by, say, id
    - Databases, as persistent systems, are extremely chatty with persistent storage/disk
    - Reading a page (or block) of data is just as expensive as reading a single item of data
    - B+ trees take advantage of this fact by loading a lot of useful data into a single block
        - Typically very large factors are chosen so that a single node will consume an entire page or block
        - Minimizes the total number of page loads required to find a particular element
    - Remember: disk is far slower than memory, and memory is far slower than the CPU
    - By using large B-Tree nodes, we minimize the interaction with the disk and maximize overall system speed
- B Trees
    - Today we looked at how B (and B+) Trees work
    - B Trees are automatically balancing trees that offer O(log(n)) operations like search, insert, delete
        - Particularly well suited for block storage technology
    - We looked at B+ Trees
        - Include all values in the leaves
        - Include “Next” pointers to make scanning fast
    - We looked at how B Trees can be used to implement database operations

Relational Algebra: Database Theory

- Welcome to my least favorite talk of the semester
- THEORY!
- Theory
    - Recall that the relational model grew out of work by E.F. Codd while at IBM
    - Codd described a set of relational algebras to provide a theoretical basis for SQL
        - Derived in many aspects from traditional Set Theory
    - What is an algebra?
        - “In its most general form, algebra is the study of mathematical symbols and the rules for manipulating these symbols”
    - Of what use is relational algebra theory?
    - I have to admit, I have never gotten much out of the topic
    - It is used in database optimization engines apparently, but we aren’t writing database optimization engines
    - Ok, so why do I teach it?
    - It’s a requirement for accreditation
    - If you continue on to 540, you will get much deeper into the theory

- We will focus mainly on the operators of relational algebra
- Relational algebra uses the term relations rather than tables
    - Slight difference between a relation and table:
        - Relations are a set
        - Tables are a bag
- What is a bag?
    - More formally called a multiset or mset
    - Allows for multiple instances for each of its elements:
    - {A, B, A, C}
    - A great programming problem is bag-equivalence between two lists… DEMO

- Relational Operators
    - Relational Algebra consists of various operators
        - Unary operators
            - Operations on single relations
        - Set operations
            - Taken directly from set theory
        - Joins and Join-like operators
            - Typically binary operators, expressing a relationship
    - Select
        - Used for selecting a subset of tuples according to a given predicate
        - Represented by the sigma symbol
        - The predicate, annoyingly, is done in subscript
        - Here we have three selects
        - “Select from tutorials where the topic is database”
        - “Select from tutorials where the topic is database and the author is ‘guru99’”
        - “Select from customers where sales are greater than 50000”
    - Project
        - Projection eliminates all attributes of the input relation but those specified in the projection list
        - Represented by the pi symbol
        - The projection list, again, annoyingly, is done in subscript
        - So here we are selecting the customer name and status “columns” from the Customers relation (table)
    - Rename
        - Rename is a unary operation (takes one argument) used for renaming attributes of a relation
        - Represented by the rho(p) symbol
        - Yep, it uses subscripts
    - Select and Project
        - From the SQL perspective these all have mappings to the SELECT statement
        - Sigma corresponds to the WHERE clause

    - Pi corresponds to the SELECT list
    - Rho corresponds to aliasing column names
- Set Operators - Union
    - UNION operators take all tuples from two relations and combines them
    - Symbolized by U symbol
    - Attribute domains must be compatible
    - Duplicates are removed
    - {A, B} U {A, C} = {A, B, C}
- Set Difference
    - Set difference operator returns all symbols in A not in B
    - Symbolized by - symbol
    - Attribute domains must be compatible
    - {A, B} - {A, C} = {B}
- Set intersection
    - Set intersection operator returns all symbols in A and in B
    - Symbolized by upside down u symbol
    - Attribute domains must be compatible
    - {A, B} (upside down u) {A, C} = {A}
- Cartesian product
    - All possible combinations of rows from relation A and relation B
    - Symbolized by X symbol
    - Attribute domains need not be compatible with one another
        - Combined with a selection it can be used as a join-like operator
    - {A, B} X {C, D} = {{A, C}, {A, D}, {B, C}, {B, D}}
- Join Operators
    - Theta join
    - General join conditional
    - Again, the conditional/predicate is expressed in a subscript
    - Here we have a relational expression on the join
        - What might this join be used for in a real database query?
- Equijoin
    - Equijoin is a specialization of the theta join
    - Uses equality only for the join condition
    - This maps to the most common join mechanism found in SQL, joining on foreign keys
- Natural Join
    - Omits the conditions
    - Implies equijioin on all compatible columns in the relations
    - Why did we say natural joins aren't used frequently in real world systems?
- Left Outer Join
    - Keeps all tuples in left relation
    - Only keeps tuples in right relation if there is a match
- Right Outer Join
    - Keeps all tuples in right relation

        - Only keeps tuples in left relation if there is a match
    - Full Outer Join
        - Keeps all tuples in both relations
        - Even tuples for which there is no match
- Relational Equations
    - Relational Equations are collections of relational operators
        - R1 is the member ID of all borrowed books naturally joined with Borrowers with book name fences
        - R2, same but book name “inheritance”
        - Final result, the intersection of these two results
- Relational Calculus
    - Another theoretical model to express conditions is with the relational calculus
    - General form is
        - { t | COND(t) }
    - We aren’t going to cover it at all beyond this mention :)
- Ok, so, what’s the point?
    - As I said earlier: I haven’t personally found a lot of use for the database theory
    - Apparently there are some results that come out of both relational calculus and relational algebra that assist with query optimization
    - We are briefly covering this there because
        - I want you to have seen the terminology in 440, so I don’t get fired
        - If you go on to 540, I don’t want you to be completely cold
- Relational Algebra
    - Today we did a quick tour of relational algebra
        - Looked at the unary operators that correspond closely to the SELECT statement
            - Select (sigma)
            - Project (pi)
            - Rename (rho)
    - Saw the notation for set operators
    - Also saw the notation for JOIN operators
    - Finally we took a brief peek at the relational calculus

Redis: Beyond DBMS

- Redis is a NoSQL data store
- Remote Dictionary Service
- Core concept is that of key-value pairs
    - Not unlike a giant hash table
- Widely used in industry
    - Almost every major website you use has redis somewhere
- Typically used as a caching layer for applications
- Here is a common cloud architecture layout
    - Application servers serve requests from users
        - First consult redis cache
        - Next consult DBMS

- Redis History
    - Redis was created by Salvatore Sanfilippo, an italian software developer
        - Aka antirez
    - Salvatore was having trouble scaling his startup with traditional DBMS systems
    - Initially written in TCL, later ported to C
    - Open sourced and announced on HackerNews in 2009
    - Quickly adopted by startups
        - Github
        - Instagram
    - Now the 4th most popular data store in the world!!!
- Popularized the idea of keeping all data in memory
    - Data can be written to disk, but only for reconstructing in memory-state
    - Initially no ACID guarantees
        - Ok, but for instagram posts, who cares about ACID?
- Features
    - Speed - Much faster than traditional DBMS
    - Persistence - sure, but not as expensive as normal DBMS
    - Data structures - API support for common data structures
        - Vs. general SQL processing
    - Atomic operations - operations on data structures are atomic but no transaction complexity
    - Replication - redis allows replication between multiple servers for failover, etc…
    - Sharding - supports sharding data amongst Redis instances
        - More on sharding later
- Redis vs. DBMS
    - Redis is awesome when
        - Data can fit in memory
        - Data doesn't need to fit into the traditional DBMS model
    - Redis is less awesome when
        - You are dealing with large amounts of data
        - You need extensive ACID guarantees around complex domain data
- Using Redis - Data Types
    - Redis data types
        - Strings
        - Hashes
        - Lists
        - Sets
        - Sorted sets
    - Internally, redis uses strings as the core data type
        - E.g. a list is a list of strings
    - Despite all primitive values being a strings Redis supports things like
        - Atomic increment
        - Atomic decrement
        - Atomic arithmetic

- Using Redis - CLI
    - Using redis is quite easy
    - Fire up a terminal and type $redis-cli
    - Should bring up the redis command line, attached to redis server running on localhost
- Basic strings
    - Setting a key value
        - Set <key><value>
    - Getting a value
        - Get <key>
    - Appending to a key value
        - Append <key><value>
    - Incrementing a value
        - Incr <key>
- Deleting a value
    - Use the delete command:
        - Del <key>
    - Value is no longer available in the redis server
- Hashes
    - Create a hash with the HSET command
        - Hset <key><key><value>
    - Retrieve with HGET
        - hget<key><key>
    - Show all values with the HGETALL command
        - Hgetall <key>
    - Remove with HDEL
        - Hdel <key><key>
- Lists
    - List commands
        - Lpush - push a value on to front of a list
    - Lpop - pop the first value off a list
    - Lrange - show the values for a given range
    - Rpush - append a value to a list
- Sets
    - Set commands
    - Sadd - add a value to a set
    - Smembers - show all members of a set
    - Sismember - 1 if element is a member of the set, 0 otherwise
    - Also supports many set operations
        - Sunion
        - Sdiff
        - Etc…
- Sorted sets
    - Sorted set commands

        - Zadd - add a value to a sorted set
        - Zrange - show all members of a sorted set, ordered low to high
        - Zrevrange - show all members of a sorted set, ordered high to low
- Java ⟷ Redis
    - To access the locally running Redis server, we will be using the Jedis client
    - Should already be imported for your application
    - Jedis has methods for Redis commands
        - Note that you are typically using Strings!
    - NB: you will need to read from and also write to this cache
        - “There are only two hard things in Computer Science: cache invalidation and naming things”
            - Phil Karlton
- Redis - Other
    - Redis has a ton of functionality
        - Can be used for synchronizing processes and threads with wait commands
        - Hyperloglog - cheap set count
        - Pub/Sub commands for more client synchronization
    - Really is an awesome piece of technology
    - One way I like to think of Redis: Online, in memory data structures
- AntiRez
    - Excellent YouTube series on writing system software
- Redis
    - A widely used NoSQL datastore
    - Uses the key value paradigm
        - Everything is stored in memory
    - Supports various types of data
        - Strings
        - Lists
        - Sets
        - Ordered sets
    - Very, very fast
    - Antirez is an absolute king

MongoDB: More NoSQL: Document Databases

- Like Redis, MongoDB is a NoSQL data store
- Unlike Redis, MongoDB is a document database
    - Stores JSON-like documents rather than offering various data types
- Documents in MongoDB
    - At right is a MongoDB document
    - As you can see, it looks a lot like JSON
    - Additionally, you can see that field values need not be a primitive type
        - Arrays
        - Nested objects
        - Etc…

    - Advantages of this approach
        - Documents (i.e. objects) correspond to native data types in many programming languages
        - Embedded documents and arrays reduce need for expensive joins
        - Dynamic schema supports fluent polymorphism
    - “Documents correspond to native data types in many programming languages”
        - To an extent
        - Obviously there is a direct mapping to JavaScript, but what about other languages?
            - Yeah, but not as good a match
            - Static typing?
    - “Embedded documents and arrays reduce need for expensive joins”
        - But what about normalization?
        - What if I want to rename the “sports” group to “college sports”?
    - “Dynamic schema supports fluent polymorphism”
        - Mmmm
        - Documents don’t need to meet any schema
        - So two documents in the same collection can look totally different…
        - Uhhhh, I guess that works for JavaScript….
    - Mongo 3.2 added support for document validation
        - Allows you to enforce a schema via JSONSchema
        - Side bar, guess who wrote this
- MongoDB Concepts
    - Database: a set of named collections
        - As well as views and a few other things
    - Collections: a set of documents
        - Stored in BSON format
- Creating Collections
    - One really nice feature of Mongo is how easy it is to create a database
    - This will create a new database, myNewDB and then create a new collection and insert some data into it
        - Very liberating if you are used to DDL
- Collections
    - Inserting multiple documents into a collection is similar, although not identical, to inserting rows in a relational database
    - You can explicitly create a collection if you wish to specify configuration options for it
        - Note the validator option
        - You can also cap the size of the collection, etc…
- Views
    - As with relational databases, you can create views in MongoDB
    - Here <source> is a query that will define the view
        - We will discuss the pipeline later
    - As with relational database, views are read only

- Documents
    - In MongoDB documents you will always have an _id field, with is of type ObjectId
        - objectIds are small, likely unique, fast to generate, and ordered. ObjectId values are 12 bytes in length, consisting of:
            - A 4-byte timestamp value, representing the objectId's creation, measured in seconds since the unix epoch
            - A 5-byte random value
            - A 3-byte incrementing counter, initialized to a random value
    - This document also shows
        - Nested objects (name)
        - Dates
        - Arrays
        - A NumberLong(64 bit integer)
            - By default the mongodb shell treats numbers as double precision floating point, like JavaScript
- CRUD in Mongo
    - Documents can be created with the insertOne() and insertMany() operations
        - This example inserts a single value into the example collection
        - Recall that the collection will automatically created if it does not exist
        - Note that you get back the object ID of the inserted document
    - Querying in Mongo is usually done with the find() method
    - Consider an inventory collection created with the following insertMany() statements
        - Properties - item, quantity, size, and status
    - A basic query using the find method takes a JSON-like query specification
    - This example finds all documents with the status "D"
    - Equivalent to the SQL at right in a relational database
    - IN-style queries use the $in keyword
    - This query returns all documents whose status is in the given list
    - Equivalent to the given SQL
    - AND queries are comma separated
    - This query returns all documents with status "A" AND a quantity less than 30
        - Note the less than specification, using the $lt keyword
        - Many comparison keywords list this are available: $eq, $gt, $gte, $in, $nin, etc…
    - OR queries use the $or keyword, which takes an array of conditions
    - Here we are finding all inventory documents with status "A" or quantity less than 30
    - ANDs and ORs can be combined with a nested query tree
    - Use a standard comma separated JSON object for AND and the $or:[] syntax for ORs
    - What about querying embedded/nested data?
    - Multiple syntaxes with slight differences
        - Obvious nested syntax requires an exact match on all fields

        - Cannot omit any fields or a document won't match
            - Note that the second query does not match any documents!
    - More intuitive syntax is available if you use the dot syntax
    - This syntax does not require a total match
    - Arrays can be queried in various ways
        - Exact match
        - Unordered match
        - Contains
        - Conditions
    - Nested documents can be queried against via the same dot syntax used for fields
    - Field selection: to select specific fields from a document, you pass in a second JSON object
        - Passing a 1 for a field name indicates it is included
        - Passing 0 for a field name indicates all other except this field should be returned
        - Can use dot syntax to include or exclude embedded documents
    - Updates are done with one of the following methods:
        - updateOne() takes a condition and an update expression and updates the first match
            - Here we update the first inventory document whose item is "paper"
            - Sets the embedded document size uom property to "cm"
            - Sets the status to "P"
            - Sets lastmodified to the current date
        - updateMany() takes a condition and an update expression and updates all matches
            - Here we update all inventory documents whose item is "paper"
            - Sets the embedded document size uom property to "in"
            - Sets the status to "P"
            - Sets lastModified to the current date
        - replaceOne() takes a condition and an update expression and replaces all matches
            - This will replace the entire document, not just the update fields
    - Upsert support
        - Any update operation can be converted into an upsert by including the {upsert : true} option
        - If no match, the document is inserted instead
        - Here, if no product with id 6 exists, the data will be inserted instead
    - Upserts are very useful in many online systems
        - Rails ORM ActiveRecord supports a similar pattern with first_or_create()
    - Delete operations
        - db.collection.deleteOne or Many(<filter>, <options>)
- Text Search
    - Mongo supports broad, google-like text search out of the box
        - First example, general text search

        - Second example: exact match for “coffee shop”
        - Third excludes “coffee”
    - Excellent functionality compared with most RDBMS
        - Especially a decade ago!
- Indexing
    - Mongo supports the easy creation of indexes
        - Supports geospatial indexing, text indexing, multi-key indexing, etc…
    - Index usage can be viewed via the $indexStats keyword
- Transactions
    - Historically Mongo has not had a good reputation for data persistence
        - Arguments over whether this is FUD or not
            - I would use an RDBMs for crucial data, myself
        - Mongo now has a transaction API
            - API is based on distributed transactions, pretty complex
- When to use Mongo?
    - As much of a technology grump as I am, I think Mongo has a place in many systems
    - Useful for:
        - Non-core data
        - Data streams
        - Flexible, early data modeling
        - Data that “doesn’t matter”
    - I would still recommend RDBMS for core, “must be correct” data
- MongoDB
    - MongoDB is a document database
        - In contrast with a relational database
        - Optimized for the storage and retrieval of documents
    - We reviewed many of the core operations for Mongo
        - CRUD
        - Indexes
    - A good SQL/Mongo mapping document
    - Next time we will talk about the aggregation pipeline in Mongo

MongoDB: Aggregation in Mongo

- Last week, we looked at MongoDB, a popular document Database
- This week we will discuss the Mongo aggregation framework
- Aggregation in SQL
    - Recall that aggregation in a RDBMS uses the GROUP BY clause in a SELECT statement
    - Groups rows of data into a single row
        - Typically paired with an aggregation function such as SUM() or COUNT()
- Aggregation in Mongo
    - Aggregation in MongoDB is similar in some ways, and distinct in others
    - Here is an example that aggregates the orders collection
    - In Mongo, aggregations are done in stages

        - In this example,
            - Stage 1 - a match stage
                - All documents matching the given filter are passed through to the next stage
            - Stage 2 - a group stage
                - All documents that passed the match stage are grouped by some set of attributes
        - Here the match stage is picking out all orders whose status is “A”
        - And the group stage is grouping by the cust_id field, and computing a sum of the amount field
        - While this may appear superficially similar to the GROUP BY clause in SQL, it is in fact more general
        - An aggregation can consist of any number of stages
        - Each stage can transform or filter documents to be passed on to the next stage
        - Stage types
            - There are many different stage types
                - $match - matches documents
                - $group - groups documents
                - $sort - sorts documents
                - $limit - limits the number of documents
                - Etc…
        - Stages can be in any order
        - Here the $group stage and the $match stage are reversed
            - This $match stage is acting like a HAVING clause in SQL: it applies to the aggregate
        - Most stage types can appear multiple times in an aggregation pipeline
            - Here we group twice
                - Once to sum up all the population by city in a zipcode database
                - And then we group that info again by the state, using an average to get the average population of cities in a state
        - The MongoDB aggregation pipeline is inspired by the Unix pipe concept
        - You connect an arbitrary number of commands together to produce the desired results
- Map/Reduce
    - Mongo also supports the Map/Reduce programming model
    - Map/Reduce is a programming model for processing large sets of data in a distributed manner
        - Cluster friendly
    - Concept has been patented by Google and was widely used there
        - As an intern I ported various batch processing jobs to the google Map/Reduce infrastructure
        - It was alright
    - According to wikipedia

        - “By 2014, Google was no longer using Map/Reduce as their primary big data processing model”
    - Nonetheless, the idea has had a large influence on the big data world
    - Many open source implementations now
        - Hadoop is the most popular one that I am aware of
    - Map/Reduce algorithm
        - Data is read as an input
        - A function then maps each piece of input to a key/value pair
        - Partition: each key/value pair is sent to a Reducer based on the mapp value
        - The reducer sorts the input and then processes it using its reduce function
        - This reduce function produces zero or more outputs
        - This output is finally written to stable storage
    - There is controversy if this is a unique or even “good” idea
    - It is taken from the functional programming concepts of the same names
    - Many distributed computing researchers felt it was not a new idea, nor particularly flexible
- Hadoop Map/Reduce
    - Hadoop infrastructure
        - Client submits a job
        - Job is split into multiple mappers
        - Mappers partition and combine data
        - Reducers (optionally) read combined data and reduce it to a final value
- Hadoop
    - Job setup
        - Configure the map reduce job
        - Submit it to the JobTracker
    - Mapper
        - Split value on tabs
        - Take the first value as year
        - Scan to end for last value
        - Parse as integer for average price for this row
        - Collect into a group based on the year
    - Reduce
        - Iterate over input grouping
        - Filter out anything with an average price less than 30
    - Not super interesting MapReduce but it gives you the flavor
    - Hadoop is very mature and widely used in industry
    - And they have a good logo
- Mongo MapReduce
    - In MongoDB, collections have a mapReduce() function available directly on them
    - This function takes three arguments
        - A map function

        - A reduce function
        - A query/output pair
    - The map function takes inputs from the collection and maps them to a particular key
    - The reduce function takes a particular key and all the values mapped to it, and transforms them to a new, final value
    - Query allows a pre-MapReduce filter to be applied to documents
    - Out allows you to specify an output collection (or inline)
- MapReduce and Hash Tables
    - MapReduce works, in part, as a sort of network aware hash table
    - Recall how a hash table works
        - A key is hashed to a particular index
        - A linked list (or similar) is maintained for each hash bucket
    - MapReduce suffers from many of the same pathological cases as Hashtables
        - E.g. What if everything maps to the same index?
- MapReduce and SQL
    - As we have discussed, aggregation in Mongo is similar but not identical to the GROUP BY functionality offered in SQL
    - This table shows a rough correspondence between SQL feature and stage type
    - Keep in mind though: Mongo allows multiple instances of most stage types
        - More like the unix pipeline than traditional SQL and RDBMS implementations
- Mongo Aggregation
    - MongoDB provides an aggregation framework similar in some ways to what is available in SQL
    - Mongo aggregation has stages rather than a fixed syntax
    - A stage has a particular stage type that provides a particular type of functionality
        - Sorting, grouping, etc…
    - Stages can be chained together, like a unix pipeline
    - Most stage types can appear more than once in a mongo aggregation
    - Mongo also provides a MapReduce API that can be used for aggregation calculations

Database Clustering: Multiple Databases

- SQLite
    - In this course we have been working with SQLite
        - A great, simple database
    - However SQLite is unique amongst major databases in that it does not provide a network interface
- Networked DBMS
    - Most other RDBMS systems (as well as NoSQL systems) offer a network interface
        - Client server architecture
    - Note that JDBC abstracts this away so java code would look very similar with another RDBMS

        - Since most databases are on the network, they aren't constrained to only talk with clients
        - They can also talk with other databases
        - Why would a database want to talk to another database?
            - Replicate data between one another?
            - Offload queries?
            - Offload other stuff?
- Clustering
    - Here we see a standard cloud architecture layout
        - Load balancers
        - Application servers (web)
        - Databases
    - These two databases can communicate with one another over the ethernet network as a "cluster"
    - Load balancer
        - Provides a single point of access to a series of servers "sitting behind it"
        - Typically very simple software, which makes it very reliable
        - Able to detect when a server "behind it" fails and re-route traffic to other working servers
    - Application server
        - Where your application logic is executed
        - Typically this is what we think of as "the app"
        - In the case of your web application, these are the servers that the Java Controllers and Model object would execute on
    - Databases
        - These are dedicated machines whose sole purpose is to run a database effectively
        - Typically have large and fast disk, as well as large memory, better CPU
        - Unlike application servers, the database needs to stay up at all times to keep the system working
            - Same with the load balancers
    - What is a cluster?
        - A cluster refers to a group of servers (in this case databases) that are connected through a network
        - A cluster typically behaves as a single resource
    - Advantages of clustering
        - High availability
            - If one db fails, another db can take its place
        - Load balancing
            - Queries and writes can be distributed across machines allowing for better performance
        - Parallel processing
            - Jobs can be split across machines to take better advantage of parallelism

        - Scalability
            - Once a cluster has been established, adding and removing a new server is relatively easy
- Clustering configurations
    - The simplest cluster setup is simply a primary and backup server
        - No traffic is routed to the backup server in normal operation
        - If the primary fails, the system moves to the backup server
    - The primary traffic between clustered servers is replication
    - Note that here the traffic is a one-way feed from the primary to the backup server
        - Simple and effective
    - No load balancing is done in this configuration however
    - To load balance a cluster requires, well, a load balancing layer as well
    - Note that all clustered servers are peers
        - This implies data replication between all nodes
        - Much more complex
            - Learn more in networking
    - We still have a single point of failure in this cluster setup: the load balancer
    - To achieve a high availability we need to remove this single point of failure
    - Now we have the previous primary/backup architecture on the load balancer level with peer clustering among servers
    - This is a high availability cluster
- MySQL clustering
    - MySQL is a popular open source database
        - Purchased and managed by Oracle
    - MySQL cluster
        - Clustering solution for MySQL
        - Originally developed by Ericsson for telecom world
    - “Shared Nothing” architecture
    - Multi-master
        - Any node in the cluster can accept writes
        - Data is partitioned redundantly across entire cluster transparently
        - Uses optimistic concurrency control for replication
    - Installation
        - Step 1
            - Set up hosts
            - Set up cluster type
            - Configure SSH key access between servers
        - Step 2
            - Configure individual host machines
            - Recall, multi-master so each node can act as its own master
        - Step 3
            - Configure “topology” of the cluster
            - Determine which nodes will fill which roles within the cluster, etc…

    - NB: cluster creation and management is typically done by a DBA or DevOps, not by developers
    - As a developer, the cluster is typically hidden from you behind an abstraction such as JDBC
- Redis Clustering
    - Redis also offers clustering
    - This redis.conf file enables clustering in an instance
        - Cluster-enable enables the clustering technology
    - Command line starts the cluster
        - 6 total servers
        - 3 master, 3 replicas
    - Connecting to the clustered redis instance will now distribute data across multiple servers
    - The data is sharded
        - We will discuss sharding in more detail in the next lecture
- MongoDB clustering
    - MongoDB has a long history of offering clustering support
    - One of the killer features early on in Mongo adoption
    - MongoDB calls servers participating in a cluster as data stores as Data Bearing Nodes
    - One data bearing node is declared primary bides
    - Other instances are secondary nodes
    - In case of a failure of a primary node, an “election” is held to promote one secondary node to the new primary node
    - Mongo clients can specify a read preference to send reads to the secondary database
        - May be faster than a read against the primary database and “good enough”
- Reporting database
    - Not typically considered a cluster, but related
    - Data is replicated from a primary database to a secondary reporting database
        - One way replication
    - Production database is tuned for high throughput
        - Indexed for application requirements
        - Expensive indexes may be omitted to maximize insert/update speeds
    - Reporting database is tuned for reporting needs
        - May be physically closer to reporting users
        - Indexed for query speed rather than insert/update
        - Long running queries that take up lots of DB resources are OK
- Data Warehouses
    - Related to the concept of reporting databases is the notion of data warehouses
    - A data warehouse is an aggregation of multiple data stores across an organization
    - Typically used in read only mode for reporting

        - ETL
            - Extract (pull data from servers)
            - Transform (standardize data)
            - Load (load into the data warehouse)
        - ETL processes and tools are a major part of the operations of large companies
- Clustering
    - Clustering is a network technology that uses multiple servers to provide
        - Redundancy
        - Parallelism
        - Scalability
    - Clustering is available in most RDBMS systems and Non-Relational DBs as well
    - Reporting databases are a similar, but distinct concept
        - Optimized for reporting needs, allow production database to optimize for report access patterns
    - Data warehouses are another similar concept, common in large companies

Sharding: Scaling Systems Horizontally

- Sharding is a technology related to, but not identical with, clustering
- Closely related to the concept of hashing/hash tables
- Hash table review
    - Recall how a hashtable works:
        - A key is chosen for the table
        - A hash function is used to compute a number
            - The hash function should be as random as possible given the input
        - The number returned from the hash function is modded by the number of buckets
        - The value is stored in that bucket
    - Note that more than one element can live in a particular bucket
    - There is not a 1-1 mapping between hash function results and items
    - This means that when you look something up in a hash table it is not just a simple bucket lookup
    - Instead, you look up the appropriate bucket and do a list scan for the key
    - What if a bucket has too many entries in it?
    - This may be due to
        - A bad hash function
        - Bad luck
        - Too many entries in the hash table
    - If the number of entries for all buckets is too high, it might be time to expand the number of buckets and rehash items
- Sharding
    - To return to sharding, here we have a simple table with a column, Column 1 that we are going to shard on
    - We compute the hash value of column 1 and then select the shard based on that value modded by the number of shards

- Logic is very close to that of hash tables
    - A “shard” = a “bucket”
- Advantages
    - Load is distributed across two shards evenly
        - Assuming a good hash function
    - Table sizes reduced
    - Index sizes reduced
    - Parallelism increased
- This is a form of horizontal scaling
    - If you want to handle more load, just add more shards
        - “Just”
    - Compare with vertical scaling
        - Increase CPU, memory, etc.. on a single machine
    - Note that sharding is a shared nothing architecture
        - Shards know nothing about other shards
- Disadvantages
    - Increased complexity of SQL
        - If you want to run queries over all rows in a table, you must consult multiple databases
    - Sharding complexity
        - You must now have sharding logic in your application layer
    - No failover mechanism
        - Sharding does not directly handle failover
    - Failover is more complex
        - If you implement failover on top of sharding, you have more complexity
    - Operational issues
        - Adding or removing columns involve multiple servers
- Despite these issues, sharding is widely used in industry
- It simply provides too much of an advantage over vertical scaling, so the complexity is worth it
    - NB: in mature systems!

- Sharding and Schemas
    - Recall our database schema
    - Which table is the best candidate for sharding in this database schema?
    - What are some problems with sharding on that table?
    - Artists are the obvious table to shard on
        - Albums and tracks are both associated with artist, so they will end up in the same shard
    - But there are problems with this
        - Other tables refer to tracks that are not tied to artists
    - What about
        - Media types and genres?
            - Maybe duplicate across all shards?

        - Playlists?
            - Hmmm, no good option
        - Invoice items
            - Yikes
    - Our schema is not a good candidate for a sharding architecture
    - However, what if we offered chinookSAAS where this functionality was provided to multiple users?
    - Now we can partition on user, and all data for a given user lives on the same shard
        - Very shard friendly
        - Very scalable
- Sharding Implementation
    - Sharding can be implemented many different ways, with two primary options
        - Application level sharding: you explicitly write sharding logic in your application code
        - Data-store level sharding: the datastore transparently shards data for you
- Rails sharding
    - Rails 6.1 introduced database sharding as a framework feature
    - Works with any backing data store
    - Can be integrated into the request setup automatically shard
        - Almost always on user
    - A lot of advantages to this approach
        - Data-store agnostic
        - Pretty simple to understand
        - Works well for simple sharding schemes
    - Disadvantages
        - Almost zero administrative help for your shards
    - This style of application level sharding can be implemented in almost any programming environment
    - Increasingly seeing libraries for doing application level sharding
- DB sharding
    - MySQL and PostgreSQL have sharding solutions but it is not baked into the core either
    - Oracle includes a sophisticated sharding architecture
        - Shard directors make sharding transparent to database connections
- Mongo sharding
    - MongoDB has integrated sharding
        - Has had sharding for a long time
        - One of the early advantages of Mongo
    - Mongo is more user friendly for sharding because it does not emphasize inter-document relationships
    - Don’t have to worry about data consistency if you don’t worry about data consistency

        - Sharding can be accomplished with a one-liner once you have the shard server configuration set up properly
        - Even I have to admit that's pretty sweet
- Redis Sharding
    - It turns out that redis automatically shards in a clustered configuration
    - This is possible because redis acts like a big hash table
        - No inter-relations between items in a redis store
- Rehashing shards
    - As with hash table buckets, shards may become too large
    - When this happens you need to increase the number of shards and rehash your data to the new, appropriate shard
        - This is a "stop the world" event
    - Google uses sharding heavily internally
    - The greatest fear (15 years ago) was "will I need to rehash my shard?"
        - Probably much better infrastructure now
- Sharding alternative
    - Sharding aims to split your data statistically, using a good hash key to distribute your data across shards
    - What if, rather than that, we assigned a user a shard number on creation?
        - Rather than computing a hash key, just use the users shard#
    - Advantages
        - Simple
        - Easier to scale horizontally
            - Just add a new server and start sending traffic to it
        - User info can be moved between shards one at a time, rather than a "stop the world" event
    - Disadvantages
        - No statistical guarantees that you are spreading your data out effectively
        - More complex key generation logic
        - Given only the non-shard user data, not possible to determine which shard they are on
- Sharding
    - Sharding is a useful technique for horizontal scaling of data stores
    - It is closely related to hash tables
    - Can be difficult to implement, depending on if your schema is shard friendly or not
    - Many systems provide sharding infrastructure
        - Application level
        - Datastore level
    - Resharding can be very expensive
    - Despite the complexity around sharding, it is widely used due to its scaling advantages

- Distributed consensus is an idea that was first discussed in the 1970s in the computer science world
- The big idea is: given a set of nodes in a distributed (i.e. networked) environment, how do we ensure they all have the same information
- In databases, this problem mainly comes up with respect to clustering
- Particularly important if you have a multi-master setup, where any node can answer any question asked of it
- Networks are notoriously chaotic and unlike, processes running within a given system, you cannot rely on
    - The other systems being up
    - The other systems being reachable
- In order to synchronize between nodes on a network, we need to be
    - Fault tolerant - a given node failing should not cause other nodes to fail as well
    - Resilient - a node that restarts should be able to rejoin the system without restarting the other nodes
- Three crucial properties of distributed consensus:
    - Termination - every “process” decides on a given value (no infinite loops)
    - Integrity - all processes must be able to compute the same value, given the same state
    - Agreement - all processes must return the same value, given the same state
- Failures:
    - Typically there are two classes of failures discussed in distributed consensus
        - “Crash” - a system stops responding and (potentially) begins responding again at a later date
        - “Byzantine” - a system may respond with incorrect data (due to compromise)
- FLP impossibility result
    - A famous paper in 1985 by fisher, lynch, and patterson proved that a deterministic algorithm for achieving distributed consensus in the presence of crashes is impossible
    - To get around this problem, distributed consensus algorithms have been developed that take advantage of randomization
    - These algorithms take advantage of randomness to ensure, in a overwhelming statistically likely case, that consensus is reached in a cluster
- PAXOS
    - Paxos is a family of protocols that solve distributed consensus problem
    - First proposed in 1989, although similar predecessors existed
    - Published in the ACM in 1998
    - Paxos algorithms are widely used in distributed systems
    - Unfortunately most of them are complicated to understand and implement
- Raft algorithm
    - The raft algorithm was designed as an alternative to paxos
    - Explicitly designed to be more easily understandable
    - Still offers the same guarantees

- Created by diego ongaro and john ousterhout at stanford university
- Published in 2013
- Now widely used in industry:
    - MongoDB
    - CockroachDB
    - RabbitMQ
    - Etc
- Main website is https://raft.github.io/
- Lots of additional resources if you are interested in this problem
- How does the raft algorithm work?
- Let's consider a simple distributed system that stores a single value, called "Term"
- In a raft based system, each node can be in one of three states:
    - Follower - is following the lead of some leader in the cluster
    - Candidate - is potentially going to be elected to the leader
    - Leader - is the leader and source of record for the rest of the cluster
- All nodes begin in the follower state
- If they do not hear from an existing leader in a given amount of time, they transition to candidate
- They then request votes from the other nodes in the cluster
- A candidate node becomes the leader if it receives votes from the majority of nodes in the cluster
- This is called the leader election in the cluster of nodes
- In this case, Node has become the leader for this cluster
- All mutations to the value "term" must now go through Node a
- A client requests that term be set to the value 5
- Node a makes a note of the change, but does not "commit" the value
- First, it must replicate the value to the follower nodes
- The leader node will only commit the value when the majority of nodes have responded that they have received the value
- The leader then notifies the other nodes that they can also commit the value
- At this point the cluster has reached consensus that the value of term should be 5
- This two step process guarantees that a majority of nodes will always agree on the value of term
- Handling failure
    - All good, but what happens when a node fails in some way?
    - To handle this, Raft has two timers, the election timeout and the heartbeat timeout
    - The election timeout
        - The election timeout is the amount of time a follower node will wait before it becomes a candidate node

        - This is effectively how long it will wait without hearing from the current leader without saying “oops, looks like the leader is gone, i’ll try to take its place”
        - This election timeout is a randomized timeout, typically between 150 and 300 milliseconds
        - Randomized so that all follower nodes do not become candidates at the same time
        - When the election timeout ends on a given node without hearing from a leader (remember, you can have multiple nodes counting down) it will become a candidate and start a new election
        - It then sends a request to other nodes for a vote
        - Other nodes will automatically vote for the new candidate unless they are also a candidate
        - The other nodes also reset their election timeout
    - The heartbeat timeout
        - Once a candidate node has a majority of votes in the cluster (which is typically quickly) it becomes the leader for the cluster
        - It begins sending append messages to the other nodes, replicating its state to them
        - The append messages serve as a way to replicate state to the new follower nodes as well as providing a heartbeat signal from the leader
        - This leader will continue being the leader until a node does not receive an append message within its heartbeat timeout
        - If a follow node does not receive a message from the leader in the heartbeat time period, it will initiate a new election
        - Whichever node wins the new election becomes the leader for the cluster
- There is a fantastic demo of the raft algorithm here:
    - Https://thesecretlivesofdata.com/raft/

CSCI 440 - The Transaction Log

- Earlier we talked about B+ trees and how they are used in databases
- They allow us to efficiently navigate and scan large amounts of data, based on a key
    - E.g. the primary key of a table
- One of the things we mentioned is good about B-Trees is that they map really well to disk
- Disks are block-based devices
- This means you get a block of data, not an individual piece of data
- Using B-Trees helps in this regard because you co-locate a lot of data on the same block
- So, for example, in the B-Tree at right, each node might occupy an entire block on disk
- DB Implementation
    - Database makers know that I/O is very expensive, so they generally cache some of the blocks (or pages) of disk in memory
    - This cache can be very complicated to manage

        - You must ensure data is written through to disk for proper ACID semantics
    - In this diagram, you can also see caches for
        - Execution plans - if the same query is executed more than once (common) don't redo your query plan
        - Connections - don't constantly close and reopen network connections
        - Results - cache results of common queries
    - All of these caches are complicated, particularly when it comes to cache invalidation
    - You can imagine what a multi-threaded nightmare this is for large database systems
- DB performance
    - Another problem that you have with databases is that writes can often lack locality
    - That is, you are often writing data to very different places on disk
    - This can make things extremely slow
    - Consider if we needed to update the records at d1 and d7
    - To do so, we would need to load in two different pages on disk and write through the changes
    - Could be extremely expensive to do so!
- To deal with these problems, database developers came up with a clever solution: the transaction log
- The big idea with the transaction log is this:
    - Rather than trying to crawl all over disk to update things, let's have a single file that we append to with transaction records
    - This is based, in turn, on the notion of "write ahead logging" (WAL)
    - Log all changes first to a log file (write ahead) then to the actual location the change applies to (a page on disk somewhere)
    - In the case of databases, the transaction log is an append-only or circular file
    - Two different transactions can update this log file in a manner that is guaranteed to not conflict
    - Note that two transactions can interleave entries in the transaction log
    - At right we see a simple transaction with a few entries in the transaction log
    - We start a transaction with the id 1
    - We update something
    - We then commit the transaction
    - Note that at this point, the update has not only made its way out to the actual file that holds the updated data
    - However, the database can tell the user that, yes, you have committed the data?
    - Why?
    - The data is persisted to disk, just not to its final location
    - What does this imply about queries that are made against the database?
    - Queries need to consult not only the pages on disk corresponding to the B-Tree associated with the table, but also the log file

        - This is where the transaction isolation level can get tricky!
        - Read uncommitted
            - Dump all deltas found in the log file into the results
        - Read committed
            - Only include deltas found in the log file
        - Repeatable read
            - Read and deltas AND lock the rows read by the query
- Transaction entries can interleave with one another
- Here some more entries have been added to the transaction log file, and you can see that transaction 2 and 3 have interleaving entries
- Because the log file is persistent
- If the power where to go out on the database machine, this file would still exist on disk
- On restart, the database could consult the log file and realize that this transaction needs to be written through to disk
- Checkpointing the Log
    - In normal operation: the Log file is periodically “checkpointed” and all entries in the log up to a certain point are written to the actual database locations
    - Here we have three transactions in flight, two of which can be “retired”
    - A “checkpoint” log entry can be added to the transaction log at this point and any entries that have already been retired by process that updates disk can be removed
    - Now those “slots” can be reused for new entries as more transaction logs are added
    - Or, if there isn’t a circular log file architecture, a new log file can be created
- Recovery with the Log
    - Log entries each have a log status number associated with them
    - These numbers are written through to the actual database entries to make recovery possible
    - Entries in the main data structure on disk can be scanned to which log status number they correspond with
    - On a restart the log can be consulted and correlated with records on disk
    - If an entry on disk has a log status number that is not associated with a commit, it can either be rolled back or rolled forward, depending on what information is available in the log
    - Checkpoints make this less expensive by guaranteeing that a database is at least synchronized to a given point in the log file
- Log entry types
    - SQLServer (from Microsoft) has the most well documented log file architecture
    - Has six different primary types of log entries
    - The entries are associated with pages rather than individual entries
    - Update log records
        - Notes an update to be made in the databases
        - Has both the before and after information about the change

        - This makes recovery much easier since you can always roll back to the state of the disk before this update occurred!
    - Compensation log records
        - Compensation log records note that a given change has been rolled back
        - As such, they are associated with one and only one update log record, the record that was rolled back
    - Commit records
        - Commit records note that a decision has been made to commit some series of update records
    - Abort records
        - Commit records note that a decision has been made to abort some series of update records
    - Checkpoint records
        - Checkpoint records note that a checkpoint has been made
        - The checkpoint record will include a reference to the first record that has not been committed to disk
    - Completion records
        - Indicates that a given transaction has been written through to disk
        - And, therefore, the entries associated with the transaction are available for removal during a checkpoint operation
- Taken all together, the log file is fundamental in ensuring that a database can properly fulfill the ACID properties
- DBAs often monitor and work directly with log files to ensure the continuing, smooth operation of their databases
- Clusters
    - Recall the simplest “clustering” setup we discussed, a primary/backup relationship
        - This isn’t a real cluster
    - What is streamed from the primary database to the backup?
        - Queries?
    - Nope, that’s a transaction log file!
    - This is much more efficient
    - The backup doesn’t need to parse SQL, plan queries, etc
    - It just needs to append the transaction log entries to its own transaction log!
    - In real clusters that are synchronized via RAFT-like algorithms, the transaction log is what is replicated
    - Much more efficient than sending raw data manipulators!
- Transaction Logs
    - So, are you going to be working directly with Transaction logs?
    - Probably not
    - But they are an important aspect of real world databases and worth having at least a passing understanding of!

CSCI 440 - Database Systems Final Review

- Course goals

        - To give you a broad understanding of relational databases
        - To help you become proficient in SQL
        - To help you be confident in schema design
        - Enable you to work with databases in code (Java)
        - Learn a bit of database theory and implementation
        - Learn about some non-relational modern tools
            - NoSQL
            - Cloud Architecture
- E/R Diagrams
    - Most commonly used E/R format today is “Crows feet”
    - Cheat sheet available here: https://www.vivekmchawla.com/erd-crows-foot-relationship-symbols-cheat-sheet/
- Visualizing a relational model
    - You should be familiar with the ChinookDB schema
- The relational model
    - Understand how foreign key references work
    - What is invoices.CustomerId encoding?
    - Why does Employees have a self-referential foreign key?
    - 1-N relationship
        - Fk is in N table
    - N-N relationship
        - Done with a join table with FK of both tables
    - 1-1 relationship
        - FK can be in either table
- Database Normalization
    - Structuring database tables such that
        - Redundancy is minimized
        - Data integrity is maximized
    - Edgar F codd: a pioneer in databases
        - Proposed “1st Normal Form” in 1970
        - Went on to propose many more increasingly strict normalized forms
    - 1st normal form
        - There is a key
        - Consider this table. Is there a key?
            - No: duplicate rows
    - 2nd normal form
        - To achieve 2NF, all data must depend on the entire key
        - The key for this table is {student, class}
        - Is there any data that depends only on part of that key?
        - Teacher depends only on class
        - To fix this, we need to pull teacher data out to a separate table
        - We are now in 2NF
        - Note that C Gross and M Wittie only appear once
            - Data redundancy has been removed

        - Easier to avoid update errors
    - 3rd Normal Form
        - 3NF demands that all data depend only on the key
        - What data here that does not depend on the key?
        - The satisfied column depends on the Grade column only
        - We now have a database in 3NF
        - It is also in BCNF
        - 3NF typically satisfies BCNF, especially with surrogate keys
        - What have we accomplished?
        - Data redundancy has been minimized
        - Update complexity has been minimized
            - E.g. it is easy to change “Satisfied” criteria now
    - Normal form summary
        - Each non-key column in a relation depends on
            - The key (1NF)
            - The whole key (2NF)
            - And nothing but the key (3NF/BCNF)
            - So help me Codd ;)
        - In the presence of a surrogate key, things become pretty obvious
            - In industry, there is always a surrogate key
        - What is the general principle here?
- Modeling Polymorphism
    - Recall your object oriented concepts:
        - Super-classes
        - Sub-classes
        - Sub-classes extend the super-class
            - Add methods and attributes
    - The relational model
        - Relations in a database are just tables and foreign keys
        - How should we model this object hierarchy if we want to store it in a table?
        - Three approaches
            - Single table inheritance
            - Class table inheritance
            - Concrete table inheritance
        - Single table inheritance
            - A single table is used for all sub-class instances
            - The columns are the union of all columns of subclasses
            - Advantages?
            - Disadvantages?
        - Class table inheritance
            - There is one table per class in the object hierarchy
            - Sub-classes include a foreign key reference to their parent classes
            - Advantages?

        - Disadvantages?
    - Concrete table inheritance
        - There is one table per concrete class in your object hierarchy
        - Advantages?
        - Disadvantages?
- Indexes
    - Indexes make queries faster by building a side data structure (a b-tree) that can be consulted when querying the database
    - What query does this index make faster?
    - What are the downsides of indexes?
- Optimistic concurrency
    - Optimistic concurrency allows concurrency issues to happen, but…
        - Reacts to them when they do
        - Assume that, for the most part, things will work out
    - Here is an implementation of optimistic concurrency, using the old value of the name field to ensure an update only occurs if the name has not been changed
    - Implementing O/C
        - Check the number of rows updated
            - If 1 row was updated, success
            - If 0 rows were updated, failure
        - On success, our optimism paid off!
        - On failure, oh well, let the user know and maybe they will try again
        - Note that this approach is very web friendly
        - If a user is looking at tickets and wanders away, no locks are being held
        - If a user accidentally picks a seat already taken (in the meantime), no worse than when using pessimistic concurrency
- What is a B-Tree?
    - Well, first off, it is obviously a Tree
        - A collection of hierarchically arranged data
    - It is also self balancing
        - No branch can get too deep compared with other branches
    - As a tree, it supports logarithmic time operations
        - Search, insert, delete
    - B-Trees are particularly well suited for block storage
        - Block storage is a storage system that consists of large chunks, or blocks, of data
        - E.g. hard drives
    - Because of this advantage, B-Trees are commonly used in databases, file systems, etc…
    - Pictured at right is a simple B Tree
    - The B tree consists of Nodes
    - Nodes hold Keys (Values) and Pointers
    - The top node is the root node
    - The bottom nodes are all leaf nodes

    - Note that a node in this b tree has room for 4 values (keys) and 5 pointers
    - The branching factor of this tree is 5
        - Note that the number of values available is always (branching factor - 1)
    - The branching factor is sometimes called the degree of the b tree
    - Because of this relationship, you will sometimes see specific B trees described as “N-M B trees”
        - N = number of keys
        - M = number of pointers
    - In our example, we have a 4-5 b-tree
- B-Tree Operations - Search
    - Search in a B Tree is similar to other tree search operations
    - Search for value 12
        - Begin in root
        - Scan numbers
        - 12 is < 16, so follow pointer
        - Scan leaf
        - Find value at second position
    - O(log(n)) - n = number of keys
- Redis
    - Redis is a NoSQL data store
    - Remote dictionary service
    - Core concept is that of key-value pairs
        - Not unlike a giant hash table
    - Widely used in industry
        - Almost every single major website you use has Redis somewhere
- MongoDB
    - Like Redis, MongoDB is a NoSQL data store
    - Unlike Redis, MongoDB is a Document Database
        - Stores JSON-like documents rather than offering various data types