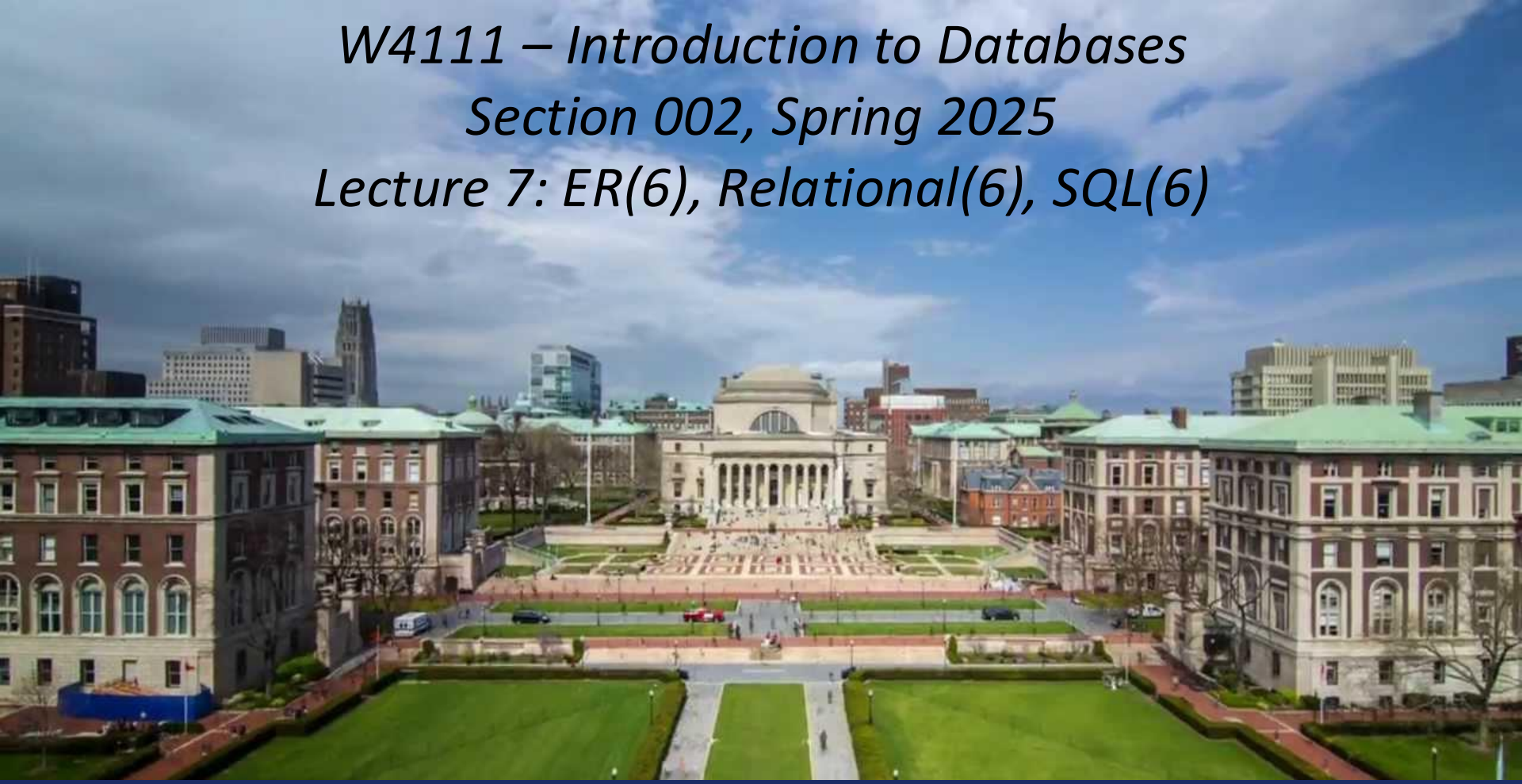
W4111 – Introduction to Databases
Section 002, Spring 2025
Lecture 7: ER(6), Relational(6), SQL(6)



Today's Contents

Contents

- Introduction and course updates.
- Finishing relational algebra. Relational algebra will be on midterm and final.
- Last topic SQL, specifically *transactions.* SQL will be part of future homework assignments, midterm and final.
- Normalization – initial overview.
- A first look at NoSQL.
- Databases and applications
  - Web applications
  - Data engineering

Introduction and Course Updates

Course Updates

- Project data set:
  - I recommend that both tracks use the IMDB and Game-of-Thrones dataset that I use in my examples.
  - But, you may choose your own dataset. I will publish requirements on your dataset tonight or tomorrow. You will describe your dataset in HW3B submission.
- HW3B:
  - Very simple. Basically, extend my examples and execute.
  - You will have to do some new DB concepts, e.g. create views.
- Midterm:
  - ~~In class, 14-MAR from 10:20 to 11:40. Please arrive by 10:10.~~
  - ~~I will cancel the 2nd half of the lecture. You may be tired and need a break from DBs.~~
  - On-line and timed (90 minutes) using Proctorio.
  - Released 14-MAR in the morning and due by 5pm.

- No one showed up for the preparatory recitation.
- 3B due after break.

Relational Algebra

Relational Algebra

- A procedural language consisting of a set of operations that take one or two relations as input and produce a new relation as their result.
- Six basic operators
  - select: σ
  - project: Π
  - union: $\cup$
  - set difference: –
  - Cartesian product: x
  - rename: ρ

What are all those other Symbols?

- τ order by
- γ group by
- $\neg$ negation
- $\div$ set division
- $\bowtie$ natural join, theta-join
- $⟕$ left outer join
- $⟖$ right outer join
- $⟗$ full outer join
- $\ltimes$ left semi join
- $\rtimes$ right semi join
- $\triangleright$ anti-join

- Some of the operators are useful and "common," but not always considered part of the core algebra.
- Some of these are pretty obscure
  - Division
  - Anti-Join
  - Left semi-join
  - Right semi-join
- Most SQL engines do not support them.
  - You can implement them using combinations of JOIN, SELECT, WHERE, ...
  - But, I cannot every remember using them in applications I have developed.
- Outer JOIN is very useful, but less common. We will cover.
- There are also some "patterns" or "terms"
  - Equijoin
  - Non-equi join
  - Natural join
  - Theta join
  - ... ...
- I may ask you to define these terms on some exams or the obscure operatorsbecause they may be common internships/job interview questions.

JOIN

- Natural Join: section ⋈ classroom
- Equivalent to

  π course_id, sec_id, semester, year,
  section.building, section.room_number, time_slot_id
  (section
  ⋈ classroom.building=section.building
  ∧
  classroom.room_number=section.room_number
  classroom)

- Equivalent to

  σ section.building=classroom.building ∧
  section.room_number=classroom.room_number
  (section × classroom)

- You can derive ⋈ from σ, π, ×
- There are naming conventions for types of JOIN
  - Natural JOIN
  - Equi-JOIN
  - Theta JOIN

LEFT, RIGHT, OUTER JOIN

-- The calculator does not like me to assign null to values.
-- So, I used 0, which violates Codd's Rules.
-- What are the R rows not in the join with NULL for T columns
--
one = π a,b,c,d←0 (R - (π a,b,c (R ⋈ T)))

-- What is "in" the join
two = R ⋈ T

-- Left out join is the union of the two.
one ∪ two

- You can derive
  - ⟕ from π, -, ∪, ⋈
  - ⟕, ⟖, ∪

Anti – Join

- "An anti-join is when you would like to keep all of the records in the original table except those records that match the other table."

instructor ▷ ID=i_id advisor

instructor.ID	instructor.name	instructor.dept_name	instructor.salary
12121	'Wu'	'Finance'	90000
15151	'Mozart'	'Music'	40000
32343	'El Said'	'History'	60000
33456	'Gold'	'Physics'	87000
58583	'Califieri'	'History'	62000
83821	'Brandt'	'Comp. Sci.'	92000

σ i_id=null (instructor ⋈ ID=i_id advisor)

instructor.ID	instructor.name	instructor.dept_name	instructor.salary	advisor.s_id	advisor.i_id
12121	'Wu'	'Finance'	90000	*null*	*null*
15151	'Mozart'	'Music'	40000	*null*	*null*
32343	'El Said'	'History'	60000	*null*	*null*
33456	'Gold'	'Physics'	87000	*null*	*null*
58583	'Califieri'	'History'	62000	*null*	*null*
83821	'Brandt'	'Comp. Sci.'	92000	*null*	*null*

Group By, Order By

classroom

classroom.building	classroom.room_number	classroom.capacity
'Packard'	101	500
'Painter'	514	10
'Taylor'	3128	70
'Watson'	100	30
'Watson'	120	50

- These are very simple examples.
- We can apply them to relations created by operations on other tables.

τ total_seats (γ building; sum(capacity)→total_seats (classroom))

classroom.building	total_seats
'Painter'	10
'Taylor'	70
'Watson'	80
'Packard'	500

Semi-Join

Semijoin (⋉)(⋊) [edit]

The left semijoin is a joining similar to the natural join and written as $R \ltimes S$ where R and S are relations.[3] The result is the set of all tuples in R for which there is a tuple in S that is equal on their common attribute names. The difference from a natural join is that other columns of S do not appear. For example, consider the tables *Employee* and *Dept* and their semijoin:

Employee

Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Production

Dept

DeptName	Manager
Sales	Sally
Production	Harriet

Employee* ⋉ *Dept

Name	EmpId	DeptName
Sally	2241	Sales
Harriet	2202	Production

More formally the semantics of the semijoin can be defined as follows:

$$R \ltimes S = \{\, t : t \in R \wedge \exists s \in S(Fun\ (t \cup s)) \,\}$$

where $Fun(r)$ is as in the definition of natural join.

The semijoin can be simulated using the natural join as follows. If a_1, ..., a_n are the attribute names of R, then

$$R \ltimes S = \Pi_{a_1,..,a_n}(R \bowtie S).$$

Since we can simulate the natural join with the basic operators it follows that this also holds for the semijoin.

Division

2.9 The **division operator** of relational algebra, "÷", is defined as follows. Let $r(R)$ and $s(S)$ be relations, and let $S \subseteq R$; that is, every attribute of schema S is also in schema R. Given a tuple t, let $t[S]$ denote the projection of tuple t on the attributes in S. Then $r \div s$ is a relation on schema $R - S$ (that is, on the schema containing all attributes of schema R that are not in schema S). A tuple t is in $r \div s$ if and only if both of two conditions hold:

- t is in $\Pi_{R-S}(r)$
- For every tuple t_s in s, there is a tuple t_r in r satisfying both of the following:

  a. $t_r[S] = t_s[S]$

  b. $t_r[R - S] = t$

- Well, that is crystal clear.
- My head hurts every time I must remember this for that one slide every semester. But, this makes a cool exam question!

Division

Division (÷) [edit]

The division is a binary operation that is written as $R \div S$. Division is not implemented directly in SQL. The result consists of the restrictions of tuples in R to the attribute names unique to R, i.e., in the header of R but not in the header of S, for which it holds that all their combinations with tuples in S are present in R. For an example see the tables *Completed*, *DBProject* and their division:

Completed

Student	Task
Fred	Database1
Fred	Database2
Fred	Compiler1
Eugene	Database1
Eugene	Compiler1
Sarah	Database1
Sarah	Database2

DBProject

Task
Database1
Database2

Completed ÷ ***DBProject***

Student
Fred
Sarah

If *DBProject* contains all the tasks of the Database project, then the result of the division above contains exactly the students who have completed both of the tasks in the Database project. More formally the semantics of the division is defined as follows:

$$R \div S = \{\ t[a_1,...,a_n] : t \in R \wedge \forall s \in S\ (\ (t[a_1,...,a_n] \cup s) \in R)\ \} \qquad (6)$$

where $\{a_1,...,a_n\}$ is the set of attribute names unique to R and $t[a_1,...,a_n]$ is the restriction of t to this set. It is usually required that the attribute names in the header of S are a subset of those of R because otherwise the result of the operation will always be empty.

Relation Algebra Summary

- There is a core set of “standard” operators in the algebra.
- Since it is an “algebra” and closed under operator composition, it is possible to “define” some additional operators that are “useful” by combining more basic operators into expressions.
- If I ask relational algebra questions on homework assignments, you can always look up the operation.
- For exams,
  - You are responsible for understanding π, σ, ←, ∪, ∩, -, ×, ⋈, theta- ⋈, ⋉, ⋊.
  - If I ask about other operators, I will give the definition.
  - I like to ask questions about how to derive operators from more basic operators.
- We are done with relational algebra in lectures. “Mic Drop.”

SQL

Using Transactions

We will cover in more detail in Module II.

SQL Parts

- DML -- provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.
- integrity – the DDL includes commands for specifying integrity constraints.
- View definition -- The DDL includes commands for defining views.
- Transaction control –includes commands for specifying the beginning and ending of transactions.
- Embedded SQL and dynamic SQL -- define how SQL statements can be embedded within general-purpose programming languages.
- Authorization – includes commands for specifying access rights to relations and views.

Transaction Concerns (I)

Alice	Bob
Read(A)	
	Read(A)
A = A-50	
	A = A-50
Write(A)	
	Write(A)

- There are 3 different programs in three different memory spaces.
  - Alice
  - Bob
  - The DBMS
- If A is initially $100
  - Alice reads $100
  - Bob reads $100
  - Alice write $50
  - Bob writes $50
- The final value of A in the DBMS is $50, but “the ATM” issued $100.

Transaction Concerns (II)

Alice
Read(A)
A = A-50
Write(A)
Read(B)
B = B+50
Write(B)

- There are 2 different programs in two different memory spaces.
  - Alice
  - The DBMS
- Programs and computers fail.
- If something fails
  - After Write(A)
  - Before Write(B)
  - The bank “lost” $50

Transaction Concept

BEGIN TRANSACTION {
1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B*)

COMMIT or ROLLBACK }

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g., transaction to transfer $50 from account A to account B:

  BEGIN TRANSACTION {
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)

  COMMIT or ROLLBACK }

  Transfer(to_a, from_b, amount)
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

Transactions

- A **transaction** consists of a sequence of query and/or update statements and is a “unit” of work
- The SQL standard specifies that a transaction begins implicitly when an SQL statement is executed.
- The transaction must end with one of the following statements:
  - **Commit work**. The updates performed by the transaction become permanent in the database.
  - **Rollback work**. All the updates performed by the SQL statements in the transaction are undone.
- Atomic transaction
  - either fully executed or rolled back as if it never occurred
- Isolation from concurrent transactions

Switch to Notebook

Normalization

The Perils of Redundancy

Features of Good Relational Designs

- Suppose we combine *instructor* and *department* into *in_dep,* which represents the natural join on the relations *instructor* and *department*

Assume that a department
- Has a single budget.
- Is in one building.

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

- There is repetition of information
  - dept_name ➔ (building, budget)
  - If I know the dept_name, I can “look up” (building, budget)
- Need to use null values (if we add a new department with no instructors)

The Perils of Redundancy

- One obvious issue is duplicate values ➔ Storage overhead/wasted space.
  - This can still be an issue for very large datasets, but … …
  - Storage has become so inexpensive that this is less of an issue.
- The primary issue is "*update anomalies.*"
  - In the schema *faculty_dept(ID, name, salary, dept_name, building, budget)*
  - The update *UPDATE faculty_dept SET building="Canary" WHERE ID='10101'* would create an anomaly.
    - 10101's dept_name is 'Comp. Sci.' but the building is not 'Taylor'.
    - This violates the logical constraint because it implies that the 'Comp. Sci.' department is in the buildings 'Taylor' and 'Canary'.
  - Relying on users and programmers to only perform "correct" updates is risky and error-prone.
  - I could prevent the problem with a *check constraint*, but the underlying issue is that the schema is "not good."
- *Normalization* formalizes what it means for a schema "to be good."

Introduction to Normalization

Features of Good Relational Designs

- Hypothetically, assume our schema originally had one relation that provided information about faculty and departments.
- Suppose we combine *instructor* and *department* into *in_dep*, which represents the natural join on the relations *instructor* and *department*

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

Instructor(id, name, salary, dept_name)
Department(dept_name, building, budget)

F:dept_name --> {building, budget}

- There is repetition of information
- Need to use null values (if we add a new department with no instructors)

Decomposition

- The only way to avoid the repetition-of-information problem in the i*n_dep* schema is to decompose it into two schemas – instructor and *department* schemas.
- Not all decompositions are good. Suppose we decompose

  employee(ID, name, street, city, salary)

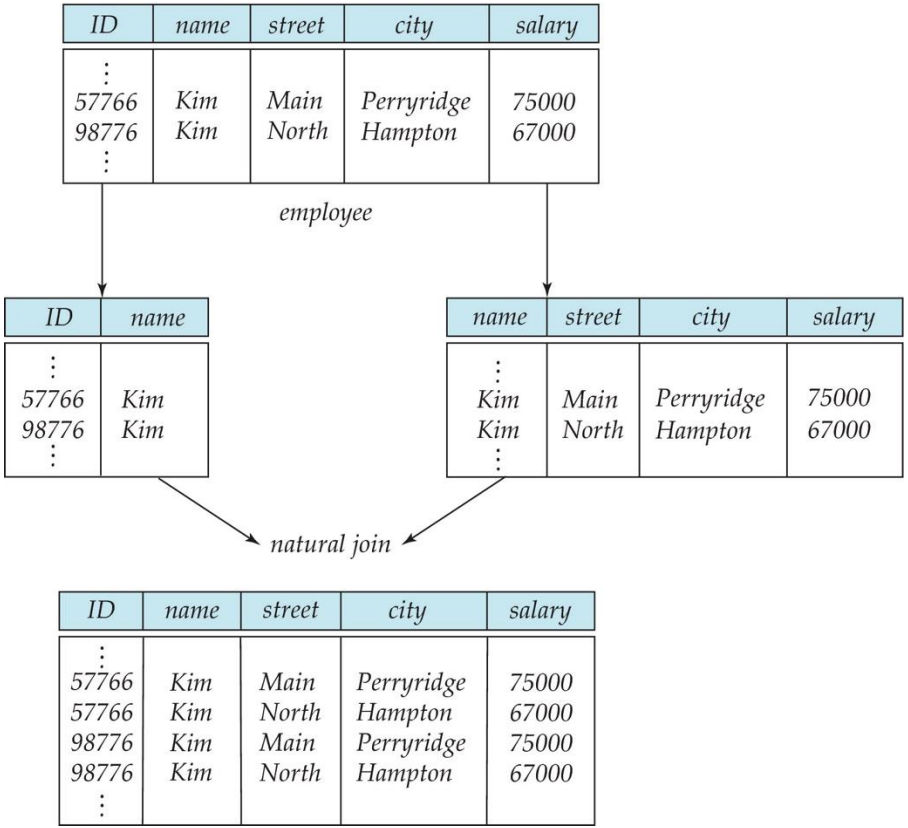
  into

  employee1 (*ID*, *name*)

  employee2 (*name*, *street, city, salary*)

  The problem arises when we have two employees with the same name
- The next slide shows how we lose information -- we cannot reconstruct the original *employee* relation -- and so, this is a **lossy decomposition**.

Lossy Decomposition



Normalization Theory

- Decide whether a particular relation *R* is in "good" form.
- In the case that a relation *R* is not in "good" form, decompose it into set of relations $\{R_1, R_2, ..., R_n\}$ such that
  - Each relation is in good form
  - The decomposition is a lossless decomposition
- Our theory is based on:
  - Functional dependencies
  - Multivalued dependencies

Functional Dependencies

- There are usually a variety of constraints (rules) on the data in the real world.
- For example, some of the constraints that are expected to hold in a university database are:
  - Students and instructors are uniquely identified by their ID.
  - Each student and instructor has only one name.
  - Each instructor and student is (primarily) associated with only one department.
  - Each department has only one value for its budget, and only one associated building.

Functional Dependencies (Cont.)

- An instance of a relation that satisfies all such real-world constraints is called a **legal instance** of the relation;
- A legal instance of a database is one where all the relation instances are legal instances
- Constraints on the set of legal relations.
- Require that the value for a certain set of attributes determines uniquely the value for another set of attributes.
- A functional dependency is a generalization of the notion of a *key.*

Functional Dependencies Definition

- Let R be a relation schema

$$\alpha \subseteq R \quad and \quad \beta \subseteq R$$

- The **functional dependency**

$$\alpha \rightarrow \beta$$

holds on R if and only if for any legal relations $r(R)$, whenever any two tuples t_1 and t_2 of r agree on the attributes α, they also agree on the attributes β. That is,

$$t_1[\alpha] = t_2[\alpha] \quad \Rightarrow \quad t_1[\beta] = t_2[\beta]$$

- Example: Consider $r(A,B)$ with the following instance of r.

1	4
1	5
3	7

- On this instance, $B \rightarrow A$ hold; $A \rightarrow B$ does **NOT** hold,

Closure of a Set of Functional Dependencies

- Given a set *F* set of functional dependencies, there are certain other functional dependencies that are logically implied by *F*.
  - If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
  - etc.
- The set of **all** functional dependencies logically implied by *F* is the **closure** of *F*.
- We denote the *closure* of *F* by F^+.

Keys and Functional Dependencies

- *K* is a superkey for relation schema *R* if and only if *K* ® *R*
- *K* is a candidate key for *R* if and only if
  - *K* ® *R*, and
  - for no a Ì *K*, a ® *R*
- Functional dependencies allow us to express constraints that cannot be expressed using superkeys. Consider the schema:

  in_dep (*ID, name, salary, dept_name, building, budget*).

  We expect these functional dependencies to hold:

  dept_name® *building*

  ID à *building*

  but would not expect the following to hold:

  dept_name ® *salary*

DFF Note:
- In the current data: ID → dept_name → building
- But
- In the schema, dept_name may be NULL..

Use of Functional Dependencies

- We use functional dependencies to:
  - To test relations to see if they are legal under a given set of functional dependencies.
    - If a relation *r* is legal under a set *F* of functional dependencies, we say that *r* **satisfies** *F.*
  - To specify constraints on the set of legal relations
    - We say that *F* **holds on** *R* if all legal relations on *R* satisfy the set of functional dependencies *F.*
- Note: A specific instance of a relation schema may satisfy a functional dependency even if the functional dependency does not hold on all legal instances.
  - For example, a specific instance of *instructor* may, by chance, satisfy
    name $\rightarrow$ *ID.*

Normal Forms

Boyce-Codd Normal Form

- A relation schema R is in BCNF with respect to a set F of functional dependencies if for all functional dependencies in F^+ of the form

  $$\alpha \rightarrow \beta$$

  where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

  - $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
  - α is a superkey for R

DFF Note:

- The theoretical treatment is not conveying the practical intent.
- If α is α superkey, I can set a primary key/unique constraint on α.
- Consider tables with address info in the rows.

Decomposing a Schema into BCNF

- Let R be a schema *R* that is not in BCNF. Let $\alpha \rightarrow \beta$ be the FD that causes a violation of BCNF.
- We decompose *R* into:
  - (α U β)
  - (*R* - (β - α))
- In our example of *in_dep*,
  - α = *dept_name*
  - β = *building, budget*

  and *in_dep* is replaced by
  - (α U β) = (*dept_name, building, budget*)
  - (*R* - (β - α)) = (*ID, name, dept_name, salary*)

DFF Note – again this is baffling

- R = (id, name_last, name_first, street, city, state, zipcode)
- α → β means zipcode → (city, state)
- (α U β) = (zipcode, city, state), which we call Address
- *R* - (β - α)). = R - β + α = (id, name_last, name_first, zipcode), which we call Person
- Setting primary key ID in Address and zipcode on Address preserves the dependency and is lossless.

Note:

- This is an example only.
- Zip code does not imply city or state in real world.

BCNF and Dependency Preservation

- It is not always possible to achieve both BCNF and dependency preservation
- Consider a schema:

  dept_advisor(s_ID, i_ID, department_name)
- With function dependencies:

  $i\_ID \rightarrow dept\_name$

  $s\_ID, dept\_name \rightarrow i\_ID$
- *dept_advisor* is not in BCNF
  - *i_ID* is not a superkey.
- Any decomposition of *dept_advisor* will not include all the attributes in

  $s\_ID, dept\_name \rightarrow i\_ID$
- Thus, the composition is NOT be dependency preserving

Third Normal Form

- A relation schema R is in **third normal form (3NF)** if for all:

  $$\alpha \rightarrow \beta \text{ in } F^+$$

  at least one of the following holds:

  - $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \in \alpha$)
  - α is a superkey for R
  - Each attribute A in $\beta - \alpha$ is contained in a candidate key for R.

    (**NOTE**: each attribute may be in a different candidate key)
- If a relation is in BCNF it is in 3NF (since in BCNF one of the first two conditions above must hold).
- Third condition is a minimal relaxation of BCNF to ensure dependency preservation (will see why later).

3NF Example

- Consider a schema:

  dept_advisor(s_ID, i_ID, dept_name)
- With function dependencies:

  $i\_ID \rightarrow dept\_name$

  $s\_ID, dept\_name \rightarrow i\_ID$
- Two candidate keys = {*s_ID, dept_name*}, {*s_ID, i_ID* }
- We have seen before that *dept_advisor* is not in BCNF
- *R,* however, is in 3NF
  - *s_ID, dept_name* is a superkey
  - $i\_ID \rightarrow dept\_name$ *and* *i_ID* is NOT a superkey, but:
    - { *dept_name*} – {*i_ID* } = {*dept_name* } and
    - *dept_name* is contained in a candidate key

Comparison of BCNF and 3NF

- Advantages to 3NF over BCNF. It is always possible to obtain a 3NF design without sacrificing losslessness or dependency preservation.
- Disadvantages to 3NF.
  - We may have to use null values to represent some of the possible meaningful relationships among data items.
  - There is the problem of repetition of information.

Normalization Simplified

Ferguson's Laws of Teaching Databases

- **First Law:** "There is no database concept so simple that database textbooks and course material cannot make the concept baffling and incomprehensible."
- **Second Law:** "No matter how baffling and incomprehensible a textbook or course material has made a simple concept, the course material will enhance the bewilderment by using unintelligible, confusing notation and quasi-math."

- My girlfriend saw this slide and added a 3rd Law.
- **Third Law: "**The grumpy professor will inevitably make the concept terrifying in addition to baffling and incomprehensible."

Goals of Normalization

- Let R be a relation scheme with a set F of functional dependencies.
- Decide whether a relation scheme R is in "good" form.
- In the case that a relation scheme R is not in "good" form, need to decompose it into a set of relation scheme $\{R_1, R_2, ..., R_n\}$ such that:
  - Each relation scheme is in good form
  - The decomposition is a lossless decomposition
  - Preferably, the decomposition should be dependency preserving.

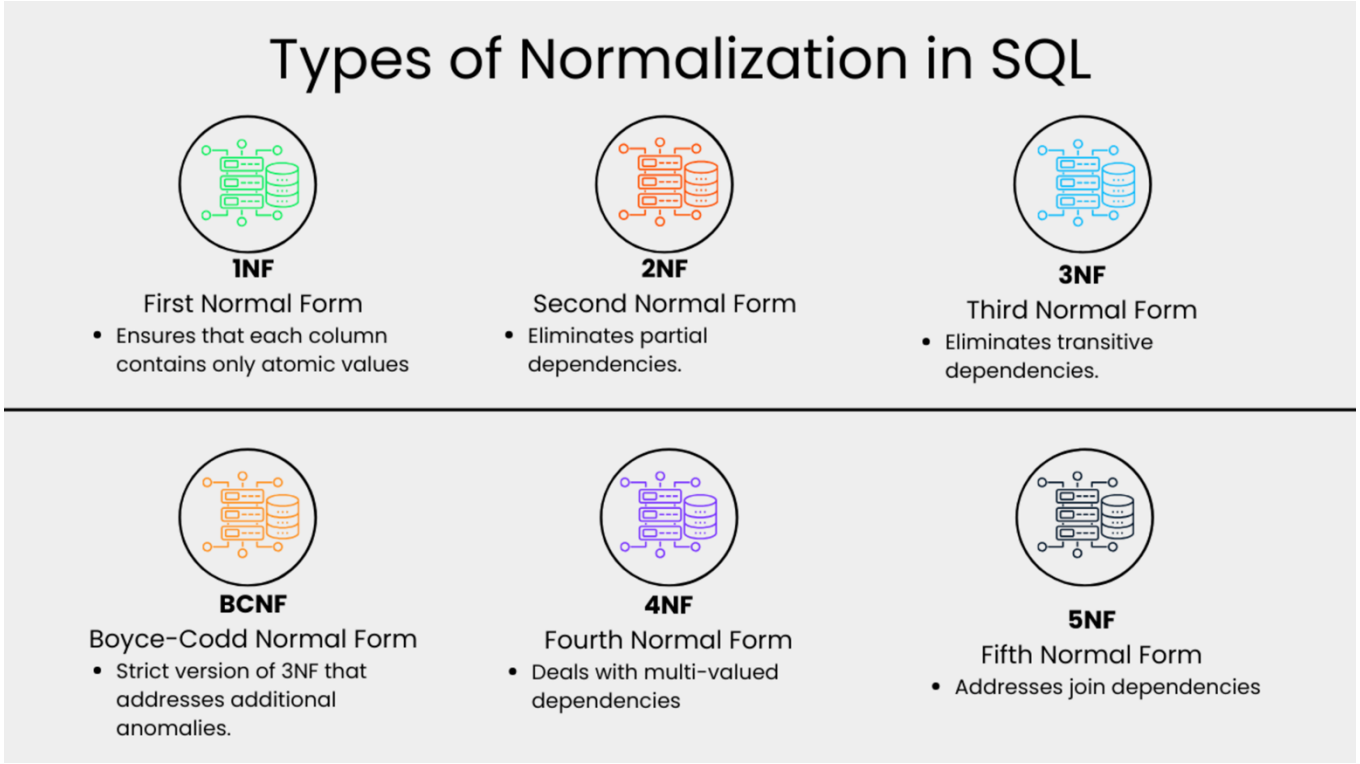
- DFF says, in a nutshell,
  - There are a few simple patterns to look for and fix.
  - Use common sense.

Denormalization for Performance

- May want to use non-normalized schema for performance
- For example, displaying *prereqs* along with *course_id,* and *title* requires join of *course* with *prereq*
- Alternative 1: Use denormalized relation containing attributes of *course* as well as *prereq* with all above attributes
  - faster lookup
  - extra space and extra execution time for updates
  - extra coding work for programmer and possibility of error in extra code
- Alternative 2: use a materialized view defined a *course* $\bowtie$ *prereq*
  - Benefits and drawbacks same as above, except no extra coding work for programmer and avoids possible errors

Simple Explanations

There are many good examples and tutorials on the web, e.g.
https://www.datacamp.com/tutorial/normalization-in-sql



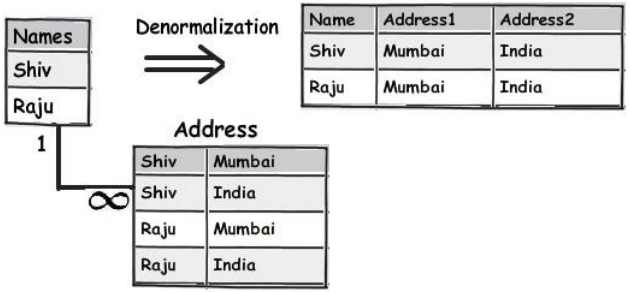
Simple Explanations

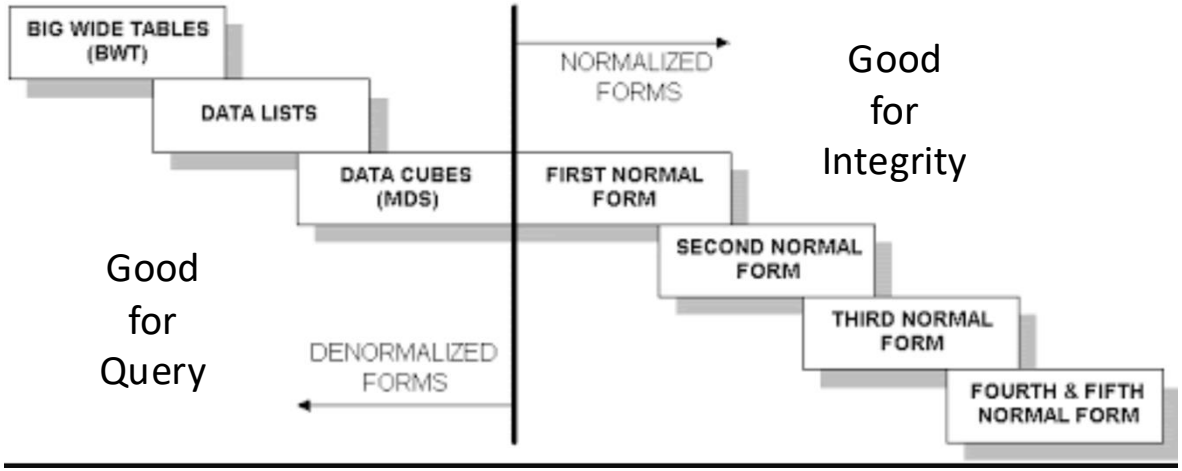
There are a lot of good, simple, short explanations and tutorials on the web.

- First Normal Form (1NF)
  - This normalization level ensures that each column in your data contains only atomic values. Atomic values in this context means that each entry in a column is indivisible. It is like saying that each cell in a spreadsheet should hold just one piece of information. 1NF ensures atomicity of data, with each column cell containing only a single value and each column having unique names.
- Second Normal Form (2NF)
  - Eliminates partial dependencies by ensuring that non-key attributes depend only on the primary key. What this means, in essence, is that there should be a direct relationship between each column and the primary key, and not between other columns.
- Third Normal Form (3NF)
  - Removes transitive dependencies by ensuring that non-key attributes depend only on the primary key. This level of normalization builds on 2NF.
- Boyce-Codd Normal Form (BCNF)
  - This is a more strict version of 3NF that addresses additional anomalies. At this normalization level, every determinant is a candidate key.
- Fourth Normal Form (4NF)
  - This is a normalization level that builds on BCNF by dealing with multi-valued dependencies.
- Fifth Normal Form (5NF)
  - 5NF is the highest normalization level that addresses join dependencies. It is used in specific scenarios to further minimize redundancy by breaking a table into smaller tables.

Wide-Flat versus Normalization

Wide Flat Tables





- Improve query performance by precomputing and saving:
  - JOINs
  - Aggregation
  - Derived/computed columns
- One of the primary strength of the relational model is maintaining "integrity" when applications create, update and delete data. This relies on:
  - The core capabilities of the relational model, e.g. constraints.
  - A well-design database (We will cover a formal definition – "normalization" in more detail later.
- Data models that are well designed for integrity are very bad for read only analysis queries.
  We will build and analyze wide flat tables as part of the analysis tasks in HW3, HW4 as projects.

NoSQL

Overview (I) (https://en.wikipedia.org/wiki/NoSQL)

A **NoSQL** (originally referring to "non SQL" or "non relational")[1] database provides a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases. Such databases have existed since the late 1960s, but did not obtain the "NoSQL" moniker until a surge of popularity in the early twenty-first century,[2] triggered by the needs of Web 2.0 companies such as Facebook, Google, and Amazon.com.[3][4][5] NoSQL databases are increasingly used in big data and real-time web applications.[6] NoSQL systems are also sometimes called "Not only SQL" to emphasize that they may support SQL-like query languages.[7][8]

Motivations for this approach include: simplicity of design, simpler "horizontal" scaling to clusters of machines (which is a problem for relational databases),[2] and finer control over availability. The data structures used by NoSQL databases (e.g. key-value, wide column, graph, or document) are different from those used by default in relational databases, making some operations faster in NoSQL. The particular suitability of a given NoSQL database depends on the problem it must solve. Sometimes the data structures used by NoSQL databases are also viewed as "more flexible" than relational database tables.[9]

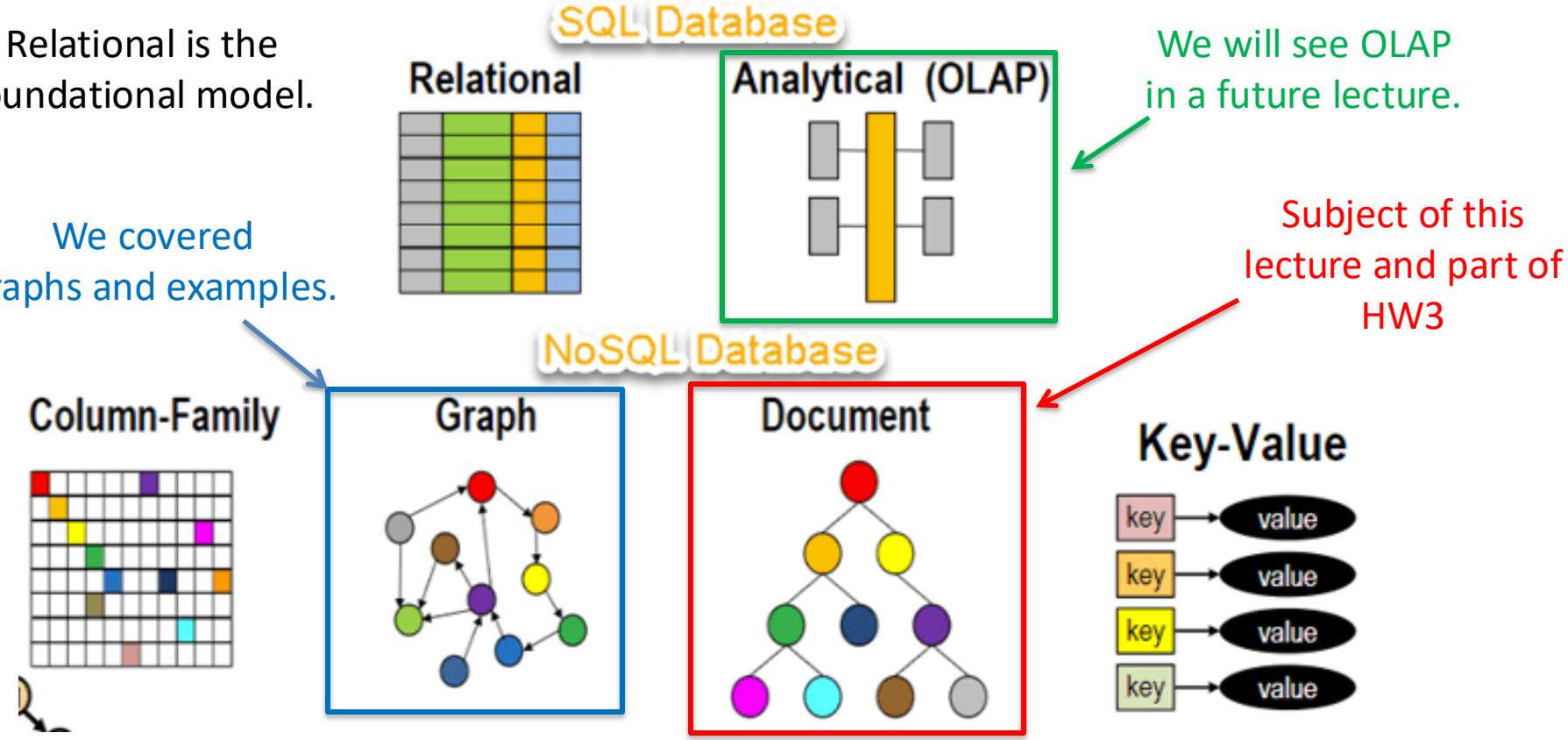
Overview (I) (https://en.wikipedia.org/wiki/NoSQL)

Many NoSQL stores compromise consistency (in the sense of the CAP theorem) in favor of availability, partition tolerance, and speed. Barriers to the greater adoption of NoSQL stores include the use of low-level query languages (instead of SQL, for instance the lack of ability to perform ad-hoc joins across tables), lack of standardized interfaces, and huge previous investments in existing relational databases.[10] Most NoSQL stores lack true ACID transactions, … …

Instead, most NoSQL databases offer a concept of "eventual consistency" in which database changes are propagated to all nodes "eventually" (typically within milliseconds) so queries for data might not return updated data immediately or might result in reading data that is not accurate, a problem known as stale reads.[11] Additionally, some NoSQL systems may exhibit lost writes and other forms of data loss.[12] Fortunately, some NoSQL systems provide concepts such as write-ahead logging to avoid data loss.[13] For distributed transaction processing across multiple databases, data consistency is an even bigger challenge that is difficult for both NoSQL and relational databases. Even current relational databases "do not allow referential integrity constraints to span databases."[14]

Simplistic Classification (https://medium.com/swlh/4-types-of-nosql-databases-d88ad21f7d3b)

Relational is the foundational model.

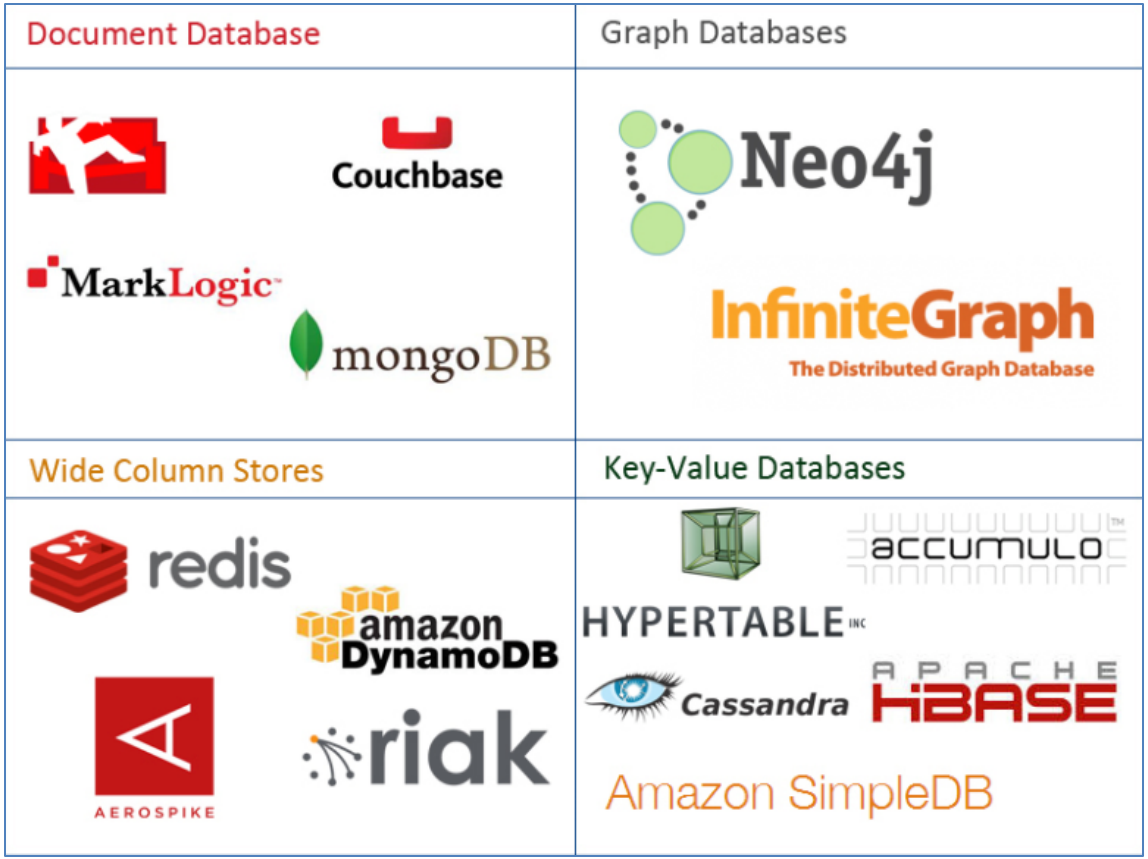


We will see OLAP in a future lecture.

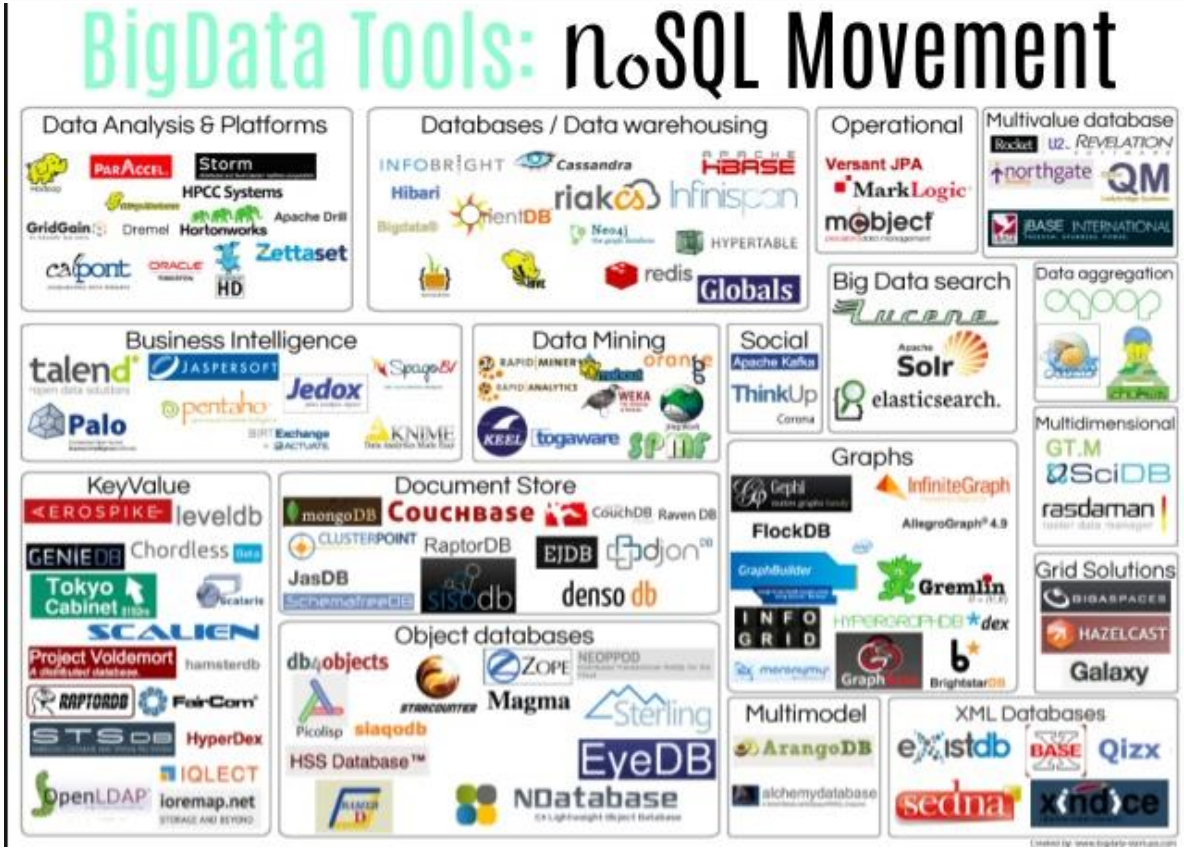
We covered graphs and examples.

Subject of this lecture and part of HW3

One Taxonomy



Another Taxonomy



Use Cases

Motivations

- Massive write performance.
- Fast key value look ups.
- Flexible schema and data types.
- No single point of failure.
- Fast prototyping and development.
- Out of the box scalability.
- Easy maintenance.

What is wrong with SQL/Relational?

- Nothing. One size fits all? Not really.
- Impedance mismatch. –
  Object Relational Mapping doesn't work quite well.
- Rigid schema design.
- Harder to scale.
- Replication.
- Joins across multiple nodes? Hard.
- How does RDMS handle data growth? Hard.
- Need for a DBA.
- Many programmers are already familiar with it.
- Transactions and ACID make development easy.
- Lots of tools to use.

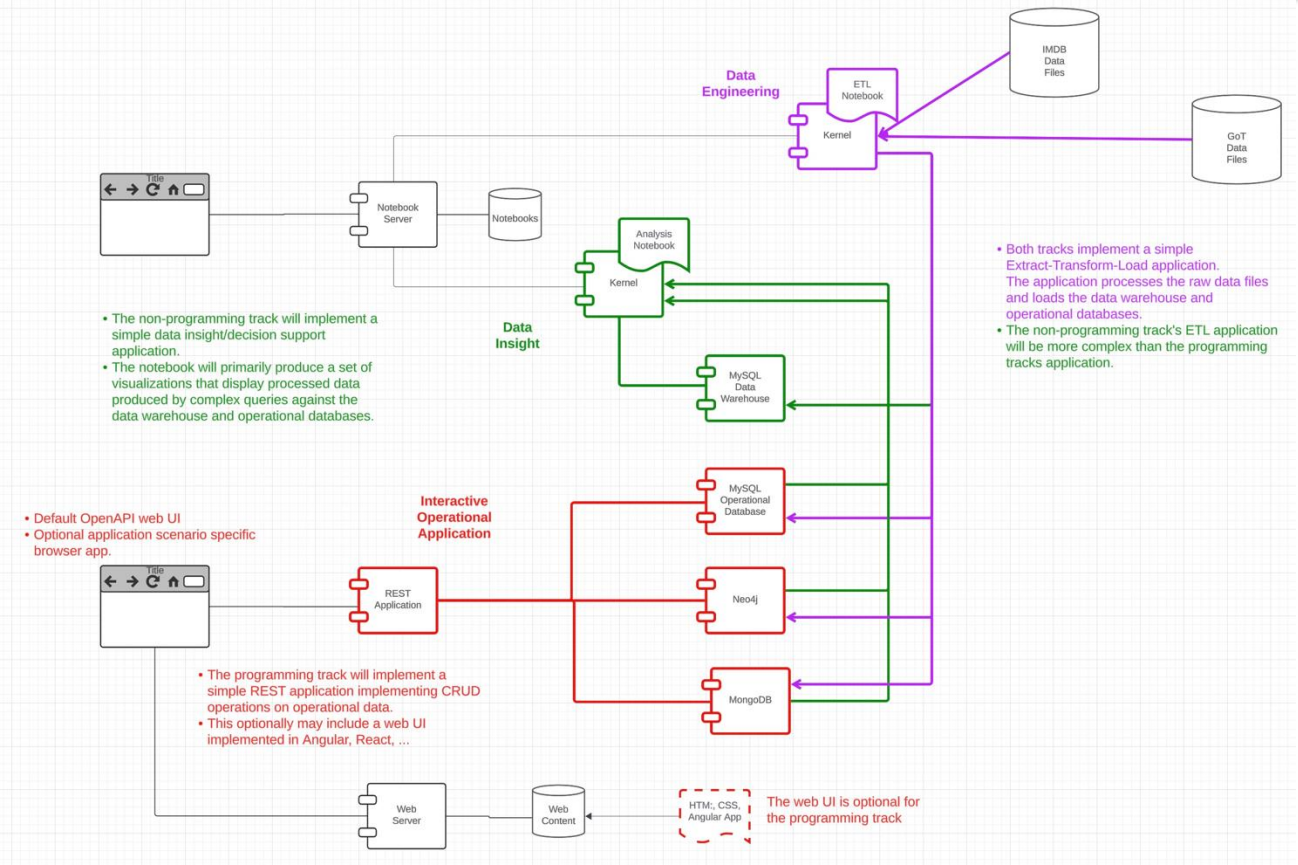
Applications

Concepts

Database Applications

- There are many, many types of database applications.
- Two common categories are
  - Operational/Interactive:
    - Users and roles can create, retrieve, update, search and delete “records.”
    - Examples: SSOL, ATMs, … …
    - Web applications are one of the more common types of operational application.
  - Business Intelligence, Decision Support, …:
    - Users can perform complex queries and analyze a lot of data to generate a report, make decisions, … …
    - Data science and AI are the fun parts; *Data Engineering* is the hard part.
    - Examples: Build AI/ML training data, dashboards, … …
- The remaining HW assignments:
  - Programming track will implement a simple web application.
  - Non-programming track will implement data engineering.

Overall System



Web Applications

Full Stack Application

Full Stack Developer Meaning & Definition

In technology development, full stack refers to an entire computer system or application from the front end to the back end and the code that connects the two. The back end of a computer system encompasses "behind-the-scenes" technologies such as the database and operating system. The front end is the user interface (UI). This end-to-end system requires many ancillary technologies such as the network, hardware, load balancers, and firewalls.

FULL STACK WEB DEVELOPERS

Full stack is most commonly used when referring to web developers. A full stack web developer works with both the front and back end of a website or application. They are proficient in both front-end and back-end languages and frameworks, as well as server, network, and hosting environments.

Full-stack developers need to be proficient in languages used for front-end development such as HTML, CSS, JavaScript, and third-party libraries and extensions for Web development such as JQuery, SASS, and REACT. Mastery of these front-end programming languages will need to be combined with knowledge of UI design as well as customer experience design for creating optimal front-facing websites and applications.

https://www.webopedia.com/definitions/full-stack/

Full Stack Web Developer

A full stack web developer is a person who can develop both **client** and **server** software.

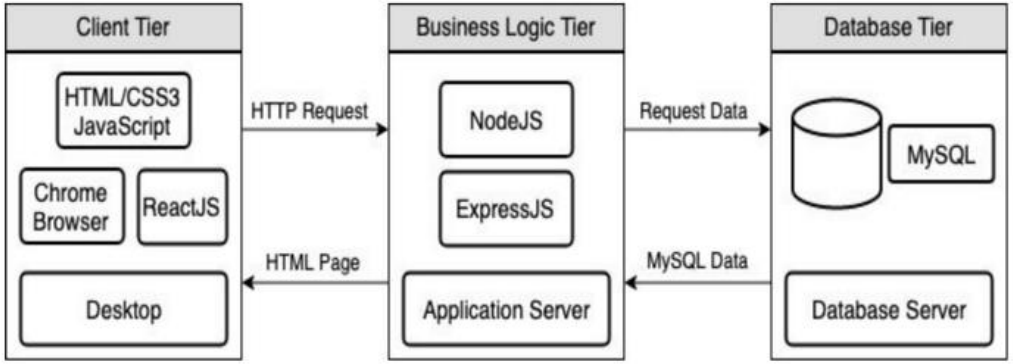
In addition to mastering HTML and CSS, he/she also knows how to:

- Program a **browser** (like using JavaScript, jQuery, Angular, or Vue)
- Program a **server** (like using PHP, ASP, Python, or Node)
- Program a **database** (like using SQL, SQLite, or MongoDB)

https://www.w3schools.com/whatis/whatis_fullstack.asp

- There are courses that cover topics:
  - COMS W4153: Advanced Software Engineering
  - COMS W4111: Introduction to Databases
  - COMS W4170 - User Interface Design
- This course will focus on cloud realization, microservices and application patterns, … …
- Also, I am not great at UIs … ... We will not emphasize or require a lot of UI work.

Full Stack Web Application



M = Mongo
E = Express
R = React
N = Node

I start with FastAPI and MySQL, but all the concepts are the same.

https://levelup.gitconnected.com/a-complete-guide-build-a-scalable-3-tier-architecture-with-mern-stack-es6-ca129d7df805

- My preferences are to replace React with Angular, and Node with Flask.
- There are three projects to design, develop, test, deploy, … ...
  1. Browser UI application.
  2. Microservice.
  3. Database.
- We will initial have two deployments: local machine, virtual machine. We will ignore the database for step 1.

Some Terms

- A web application server or web application framework: "A web framework (WF) or web application framework (WAF) is a software framework that is designed to support the development of web applications including web services, web resources, and web APIs. Web frameworks provide a standard way to build and deploy web applications on the World Wide Web. Web frameworks aim to automate the overhead associated with common activities performed in web development. For example, many web frameworks provide libraries for database access, templating frameworks, and session management, and they often promote code reuse." (https://en.wikipedia.org/wiki/Web_framework)

- REST: "REST (Representational State Transfer) is a software architectural style that was created to guide the design and development of the architecture for the World Wide Web. REST defines a set of constraints for how the architecture of a distributed, Internet-scale hypermedia system, such as the Web, should behave. The REST architectural style emphasises uniform interfaces, independent deployment of components, the scalability of interactions between them, and creating a layered architecture to promote caching to reduce user-perceived latency, enforce security, and encapsulate legacy systems.[1]

  REST has been employed throughout the software industry to create stateless, reliable web-based applications." (https://en.wikipedia.org/wiki/REST)

Some Terms

- OpenAPI: “The OpenAPI Specification, previously known as the Swagger Specification, is a specification for a machine-readable interface definition language for describing, producing, consuming and visualizing web services.” (https://en.wikipedia.org/wiki/OpenAPI_Specification)

- Model: “A model represents an entity of our application domain with an associated type.” (https://medium.com/@nicola88/your-first-openapi-document-part-ii-data-model-52ee1d6503e0)

- Routers: “What fastapi docs says about routers: If you are building an application or a web API, it’s rarely the case that you can put everything on a single file. FastAPI provides a convenience tool to structure your application while keeping all the flexibility.” (https://medium.com/@rushikeshnaik779/routers-in-fastapi-tutorial-2-adf3e505fdca)

- Summary:
  - These are general concepts, and we will go into more detail in the semester.
  - FastAPI is a specific technology for Python.
  - There are many other frameworks applicable to Python, NodeJS/TypeScript, Go, C#, Java, … …
  - They all surface similar concepts with slightly different names.

REST (https://restfulapi.net/)

What is REST

- REST is acronym for **RE**presentational **S**tate **T**ransfer. It is architectural style for **distributed hypermedia systems** and was first presented by Roy Fielding in 2000 in his famous dissertation.
- Like any other architectural style, REST also does have it's own 6 guiding constraints which must be satisfied if an interface needs to be referred as **RESTful**. These principles are listed below.

Guiding Principles of REST

- **Client–server** – By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components.
- **Stateless** – Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client.
- **Cacheable** – Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.
- **Uniform interface** – By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components. REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state.
- **Layered system** – The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond the immediate layer with which they are interacting.
- **Code on demand (optional)** – REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented.

Resources

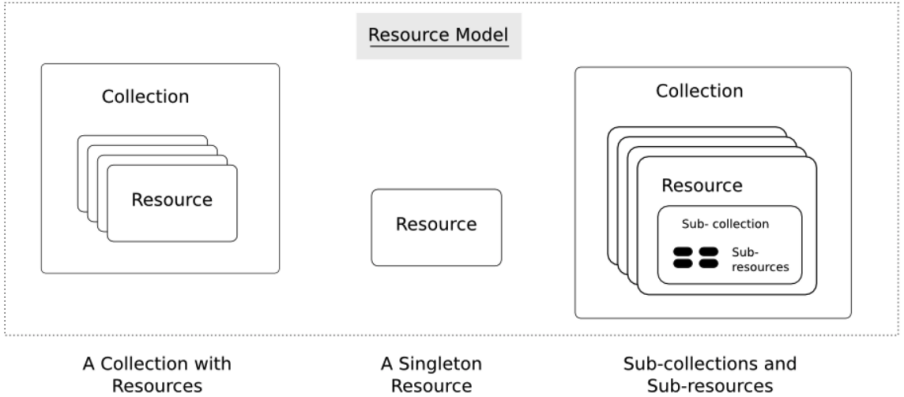
Resources are an abstraction. The application maps to create things and actions.

"A resource-oriented API is generally modeled as a resource hierarchy, where each node is either a *simple resource* or a *collection resource*. For convenience, they are often called a resource and a collection, respectively.

- A collection contains a list of resources of **the same type**. For example, a user has a collection of contacts.
- A resource has some state and zero or more sub-resources. Each sub-resource can be either a simple resource or a collection resource.

For example, Gmail API has a collection of users, each user has a collection of messages, a collection of threads, a collection of labels, a profile resource, and several setting resources.

While there is some conceptual alignment between storage systems and REST APIs, a service with a resource-oriented API is not necessarily a database, and has enormous flexibility in how it interprets resources and methods. For example, creating a calendar event (resource) may create additional events for attendees, send email invitations to attendees, reserve conference rooms, and update video conference schedules. (Emphasis added) (https://cloud.google.com/apis/design/resources#resources)



https://restful-api-design.readthedocs.io/en/latest/resources.html

REST – Resource Oriented

- When writing applications, we are used to writing functions or methods:
  - openAccount(last_name, first_name, tax_payer_id)
  - account.deposit(deposit_amount)
  - account.close()

  We can create and implement whatever functions we need.
- REST only allows four methods:
  - POST: Create a resource
  - GET: Retrieve a resource
  - PUT: Update a resource
  - DELETE: Delete a resource

  That's it. That's all you get.

"The key characteristic of a resource-oriented API is that it emphasizes resources (data model) over the methods performed on the resources (functionality). A typical resource-oriented API exposes a large number of resources with a small number of methods." (https://cloud.google.com/apis/design/resources)

- A REST client needs no prior knowledge about how to interact with any particular application or server beyond a generic understanding of hypermedia.

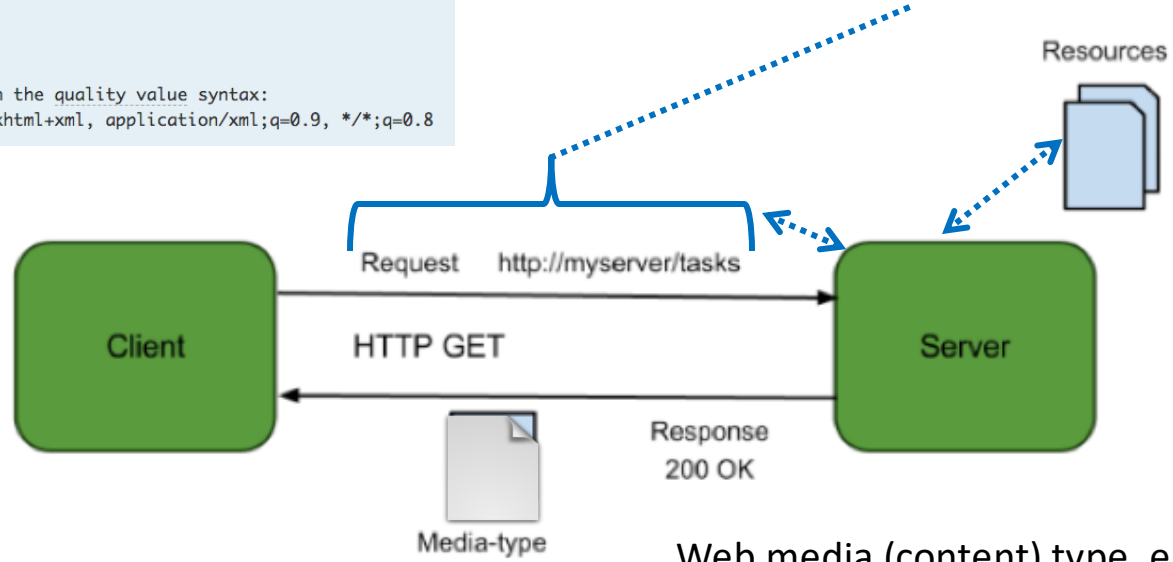
Resources, URLs, Content Types

Accept type in headers.

```
Accept: <MIME_type>/<MIME_subtype>
Accept: <MIME_type>/*
Accept: */*

// Multiple types, weighted with the quality value syntax:
Accept: text/html, application/xhtml+xml, application/xml;q=0.9, */*;q=0.8
```

- Relative URL identifies “resource” on the server.
- Server implementation maps abstract resource to tangible “thing,” file, DB row, … and any application logic.



Client may be

Browser

Mobile device

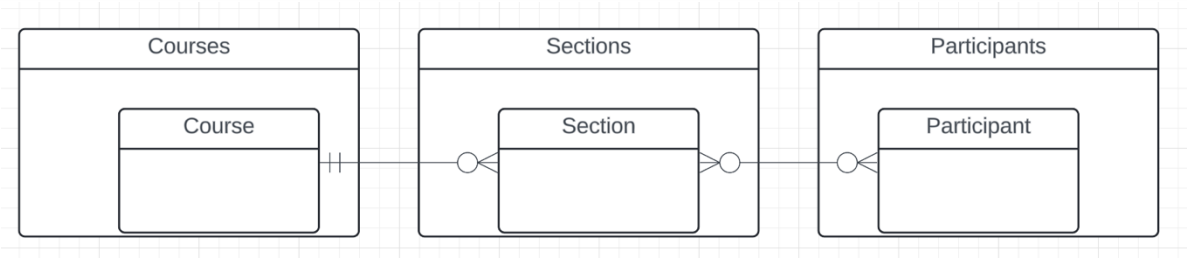
Other REST Service

… …

Web media (content) type, e.g.

- text/html
- application/json

Resources and APIs



- Base resources, paths and methods:
  - /courses: GET~~, POST~~
  - /courses/<id>: GET, ~~PUT, DELETE~~
  - /sections: GET, ~~POST~~
  - /sections/<id>: GET, ~~PUT, DELETE~~
  - /participants: GET, ~~POST~~
  - /participants/<id>: GET, ~~PUT, DELETE~~
- There are relative, navigation paths:
  - /courses/<id>/sections
  - /participants/<id>/sections
  - etc.
- There are two approaches to defining API
  - Start with OpenAPI document and produce an implementation template.
  - Start with annotated code and generate API document.
- In either approach, I start with *models.*
- Also,
  - I lack the security permission to update CourseWorks.
  - I can choose to not surface the methods or raise and exception.
- GET on resources that are collections may also have query parameters.

Data Modeling Concepts and REST

Almost any data model has the same core concepts:

- Types and instances:
  - Entity Type: A definition of a type of thing with properties and relationships.
  - Entity Instance: A specific instantiation of the Entity Type
  - Entity Set Instance: An Entity Type that:
    - Has properties and relationships like any entity, but ...
    - Has at least one *special relationship* – ***contains.***
- Operations, minimally CRUD, that manipulate entity types and instances:
  - Create
  - Retrieve
  - Update
  - Delete
  - Reference/Identify/... ...
  - Host/database/table/pk

REST (https://www.tutorialspoint.com/restful/restful_introduction.htm)

What is REST architecture?

REST stands for REpresentational State Transfer. REST is web standards based architecture and uses HTTP Protocol. It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods. REST was first introduced by Roy Fielding in 2000.

In REST architecture, a REST Server simply provides access to resources and REST client accesses and modifies the resources. Here each resource is identified by URIs/ global IDs. REST uses various representation to represent a resource like text, JSON, XML. JSON is the most popular one.

HTTP methods

Following four HTTP methods are commonly used in REST based architecture.

- **GET** – Provides a read only access to a resource.
- **POST** – Used to create a new resource.
- **DELETE** – Used to remove a resource.
- **PUT** – Used to update a existing resource or create a new resource.

REST (https://www.tutorialspoint.com/restful/restful_introduction.htm)

Introduction to RESTFul web services

A web service is a collection of open protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer. This interoperability (e.g., between Java and Python, or Windows and Linux applications) is due to the use of open standards.

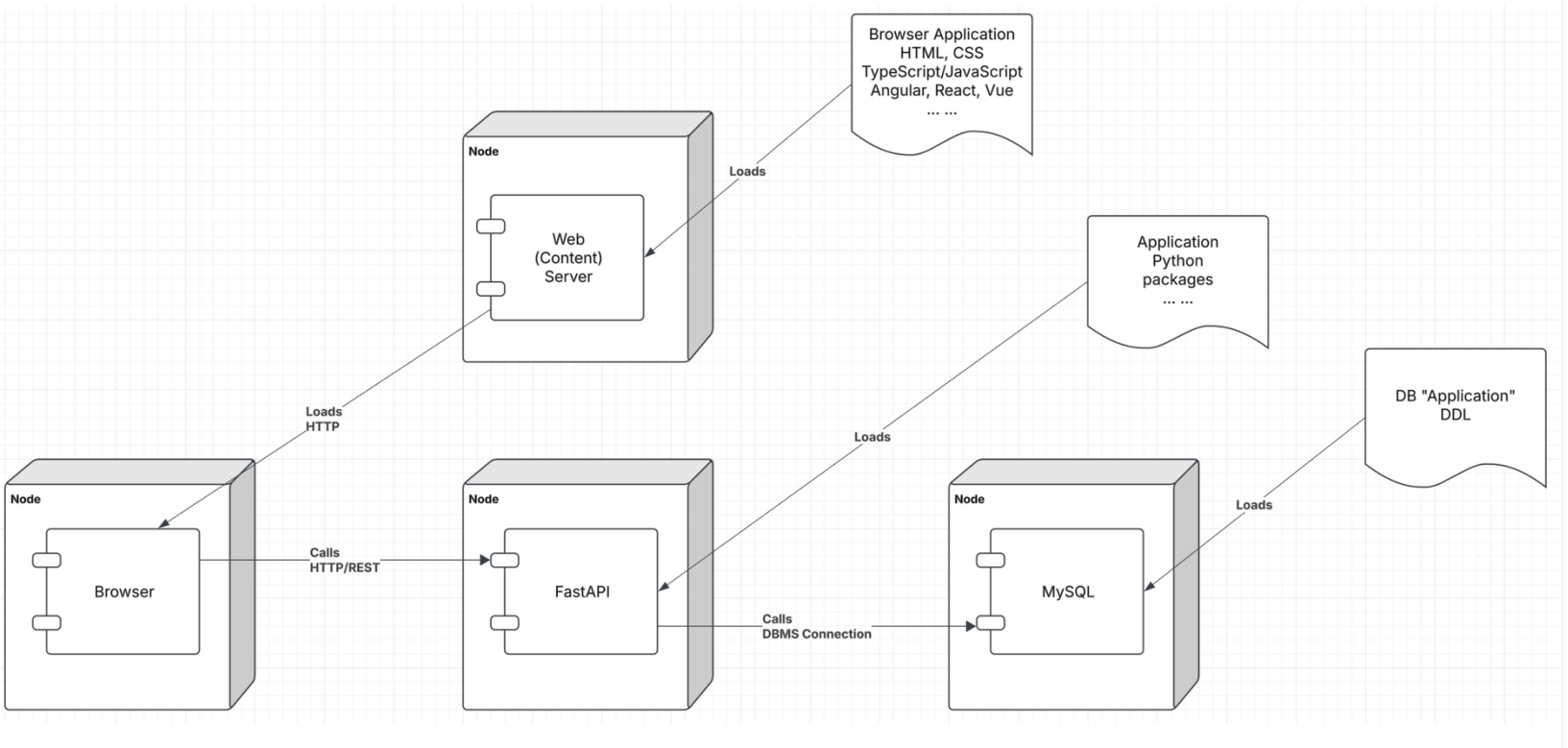
Web services based on REST Architecture are known as RESTful web services. These webservices uses HTTP methods to implement the concept of REST architecture. A RESTful web service usually defines a URI, Uniform Resource Identifier a service, provides resource representation such as JSON and set of HTTP Methods.

Creating RESTFul Webservice

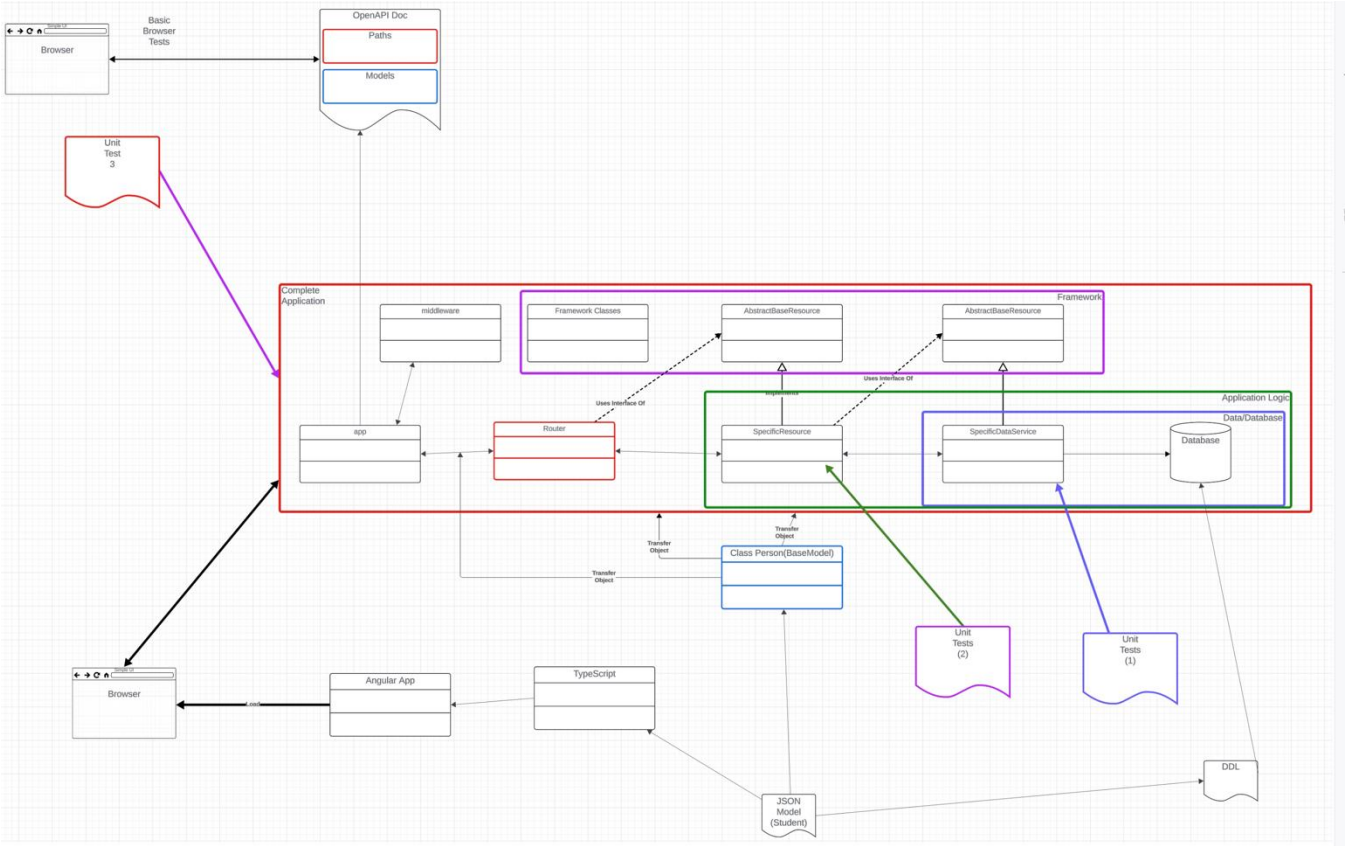
In next chapters, we'll create a webservice say user management with following functionalities –

Sr.No.	URI	HTTP Method	POST body	Result
1	/UserService/users	GET	empty	Show list of all the users.
2	/UserService/addUser	POST	JSON String	Add details of new user.
3	/UserService/getUser/:id	GET	empty	Show details of a user.

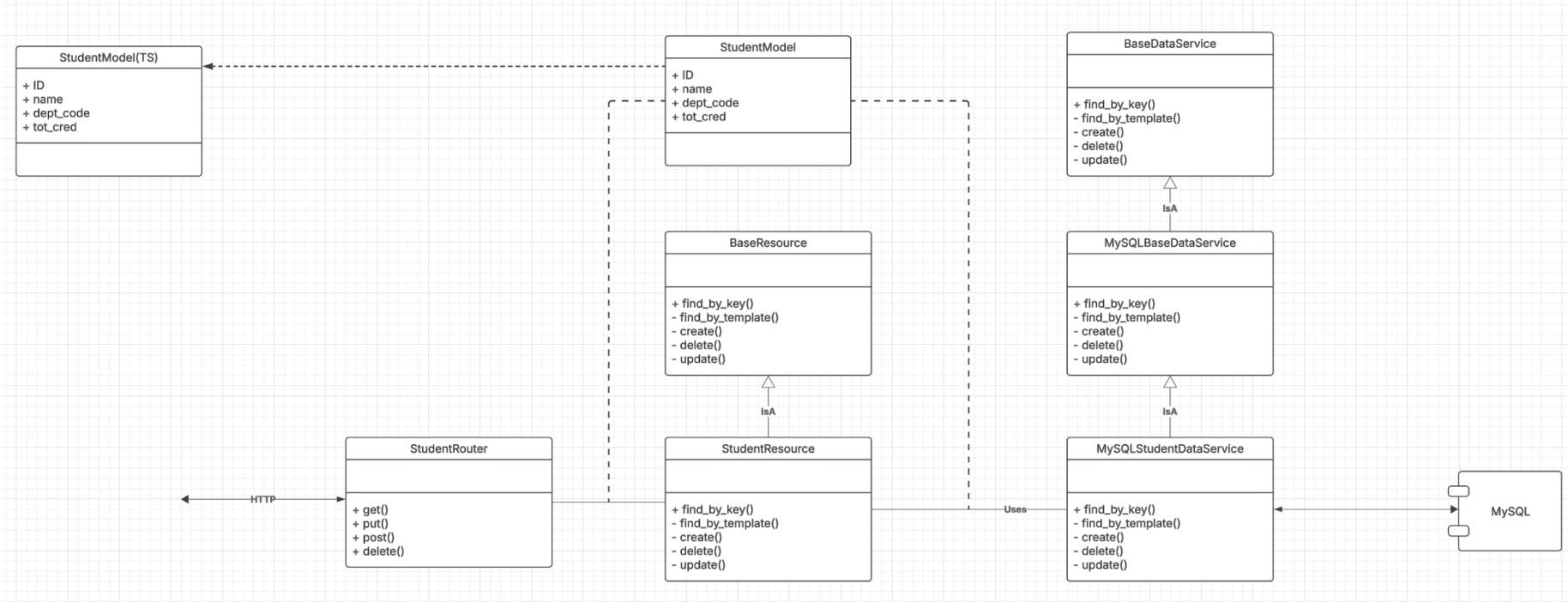
Let’s Look at the Programming Project



Let’s Look at the Programming Project



Let’s Look at the Programming Project

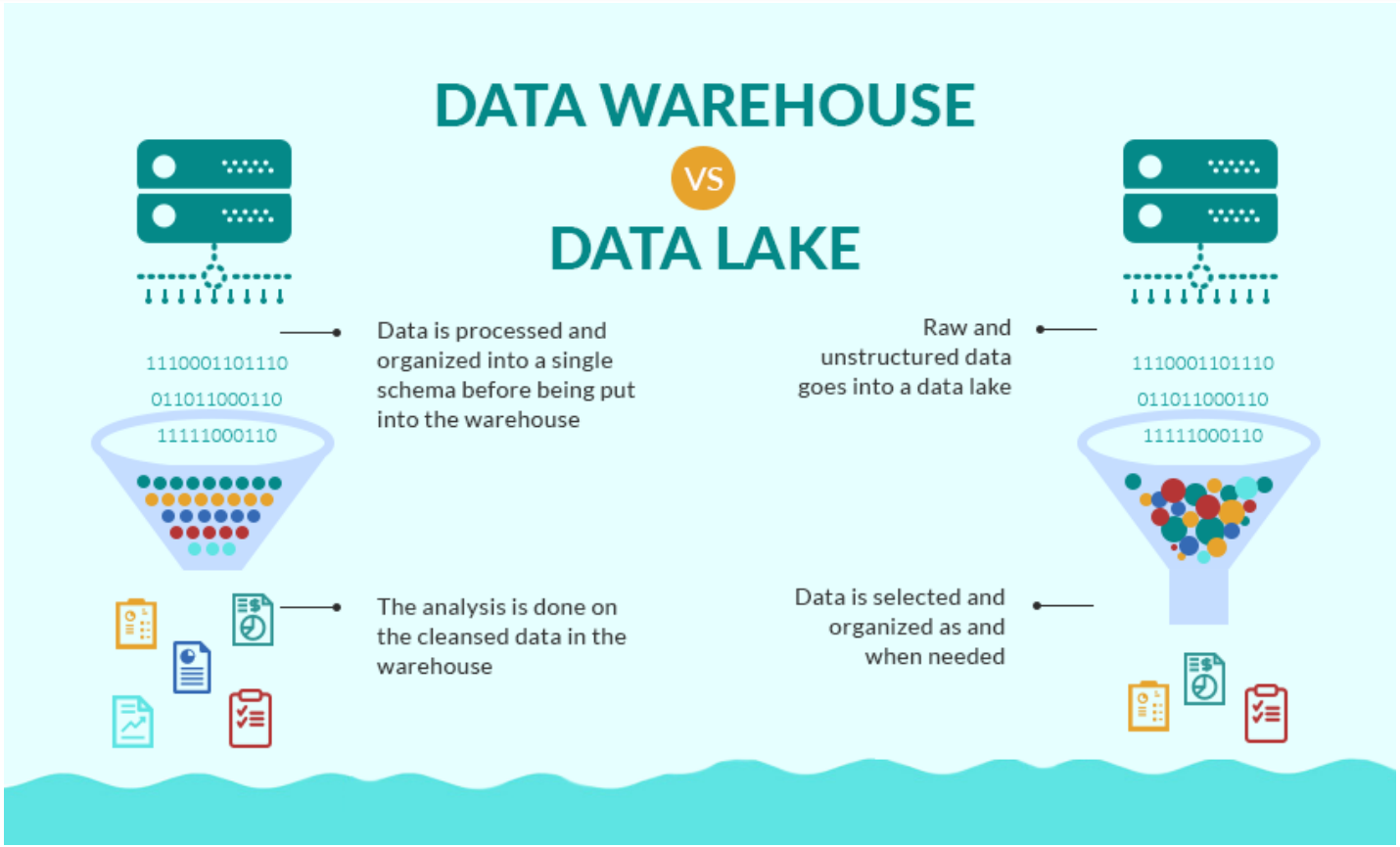


A Simple Example

- Let's assume that we wanted to extend the template to surface
  - db_book.instructor and db_book.department
  - As resources.
  - This would result in two resources *instructor* and *department.*
- The first steps are:
  1. Create *models* for *instructor* and *department*.
  2. Create *DepartmentMySQLDataService* and *InstructorMySQLDataService.*
  3. Unit test, using the functionality of the base class.
  4. Make *DepartmentResource* and *InstructorResource* by copying and modifying *StudentResource.*
  5. Follow the pattern in service_factory.py to set up the services and resources.
  6. Copy and modify interactive_app/ts/t_artist_resource.py and unit test.
  7. Modify main.py to have the necessary imports.
  8. Add the paths for /instructor, /department, /instructor/{ID}, /department/{dept_name}
  9. Execute
  10. Go to docs to test.

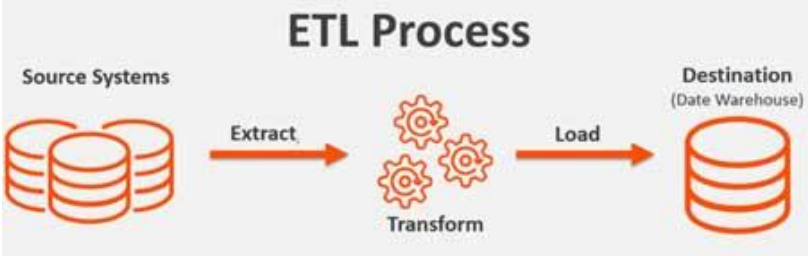
Data Engineering

Data Warehouse and Data Lake



ETL Concepts

https://databricks.com/glossary/extract-transform-load



Extract

The first step of this process is extracting data from the target sources that could include an ERP, CRM, Streaming sources, and other enterprise systems as well as data from third-party sources. There are different ways to perform the extraction: **Three Data Extraction methods:**

1. Partial Extraction – The easiest way to obtain the data is if the if the source system notifies you when a record has been changed
2. Partial Extraction- with update notification - Not all systems can provide a notification in case an update has taken place; however, they can point those records that have been changed and provide an extract of such records.
3. Full extract – There are certain systems that cannot identify which data has been changed at all. In this case, a full extract is the only possibility to extract the data out of the system. This method requires having a copy of the last extract in the same format so you can identify the changes that have been made.

Transform

Next, the transform function converts the raw data that has been extracted from the source server. As it cannot be used in its original form in this stage it gets cleansed, mapped and transformed, often to a specific data schema, so it will meet operational needs. This process entails several transformation types that ensure the quality and integrity of data; below are the most common as well as advanced transformation types that prepare data for analysis:

- Basic transformations:
- Cleaning
- Format revision
- Data threshold validation checks
- Restructuring
- Deduplication
- Advanced transformations:
- Filtering
- Merging
- Splitting
- Derivation
- Summarization
- Integration
- Aggregation
- Complex data validation

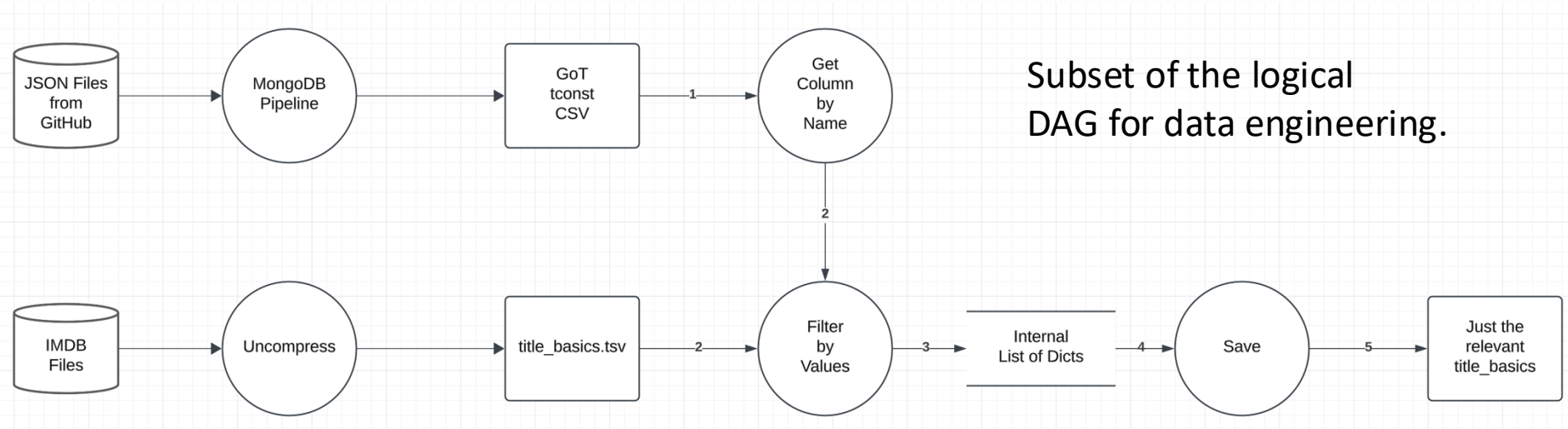
Load

Finally, the load function s is the process of writing converted data from a staging area to a target database, which may or may not have previously existed. Depending on the requirements of the application, this process may be either quite simple or intricate.

Data Engineering Overview

- We will spend 1.5 to 2 lectures in Module IV on “Big Data,” and this will cover data engineering. ➔ We are not going to spend time explaining it now.
- The core concepts are
  - Extract – Transform – Load aka Extract – Load – Transform
  - Data Warehouse, Data Lake
  - Directed Acyclic Data Flow Graphs
  - Algebraic Transformation Operators
- We are going to keep this simple for now:
  - MySQL will be the data warehouse.
  - We will use Jupyter notebooks and SQL for ETL.
- Go to data engineering example in project template.

Simple DAG



Subset of the logical DAG for data engineering.

- Data sources to extract:
  - IMDB: https://developer.imdb.com/non-commercial-datasets/
  - Game of Thrones: https://github.com/jeffreylancaster/game-of-thrones
- Sample implementation in project template: common_data_engineering/Data_Engineering_Phase_1_Example.ipynb