Problems with using flat files:

- data integrity and redundancy
- sysadmin is the bottleneck
- lack of atomicity
- concurrent access
- security

Database: abstracts away how data is stored, maintained, and processed

- big data and distributed systems have reintroduced the importance of understanding how data is laid out on disks/nodes

A database:

- Provides way to Create, Read, Update, Delete (CRUD) data without worrying about files and breaking data integrity
- provides one single location for all data in the database
- is little more than specialized data structures on disk/SSD, some NoSQL systems like ReDiS (Remote Dictionary Server) are called “data structure servers”
    - Data structures: hash tables, trees

Database-like systems:

- blockchain is similar to distributed database, where each participant maintains their own data and updates to the database. All nodes in the system cooperate to make sure the database comes to the same conclusion, a form of security
    - network of users acts as a consensus mechanism
- git: stupid content tracker–git can be used to track any kind of content

Levels of abstraction:

- physical: how data is stored on disk and accessed
- logical: how we read and write data
- view: use case specific

Logical abstraction:

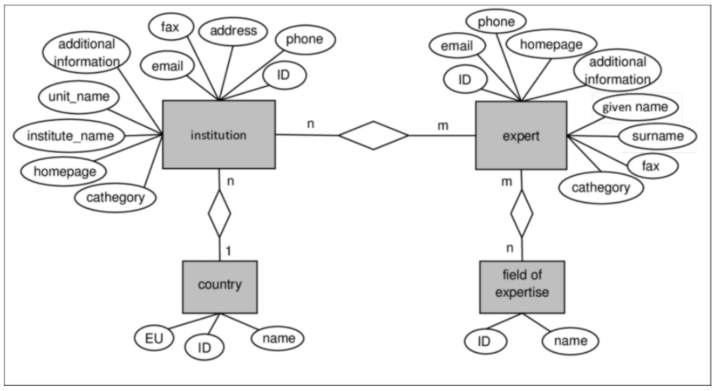
- way of abstracting a structured type–describes what data is in the database and how they are represented, as well as relationships among data
    - e.g.

```
struct Section {
        int srs;
        int cap;
        int instructor_uid;
        int parent_srs;
}
```

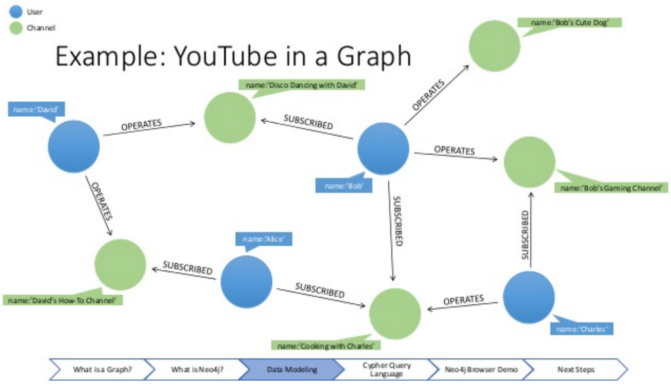
View level abstraction:

- highest level of abstraction–provides only the data needed for a particular use case
    - e.g. given instructor data, an email directory use case only requires name and email, not all instructor data such as UID and SSN

- The information stored in a database at a particular point of tim eis called an **instance** (some state of a database)
- **Schema**: overall design of a database
    - logical structure of a database or a relation
- **Data model**: defines how we design databases or interact with data
    - How do we define data?

- How do we encode relationships among data?
- How do we impose constraints on data? (e,g, data types, data integrity)
- 5 major types of models based on implementation (actually how data is stored on disk) and design mechanisms (abstraction for how data is stored)
    - **relational (implementation)**: data stored as a relation. Rows (records) represent individual units called tuples, columns represent attributes common to all records and have specific types
        - e.g. MySQL, PostgreSQL, SQlite, Oracle, MS SQLServer, DB2
    - **entity-relationship (ER) (design mechanism)**: uses a collection of basic objects called entities and relationships among them. WE typically use this to visualize a database design
        - 

            - explicit cardinality
            - relationships are explicitly defined
    - **object-oriented (design mechanism)**: draws a strong analogy to object-oriented programming with encapsulation, methods, object identity
        - ORM: object relational model
            - e.g. Django, SQLAlchemy, PHP Laravel
    - **document/semi-structured (implementation?)**: individual data objects may have different sets of attributes:
        - records may not share a core set of attributes, each record can have its own attributes
        - JSON and XML are two examples of data types for semi-structured model
        - e.g. MongoDB, Firebase, DynamoDB, CouchDB
    - **network/hierarchical/graph (implementation)**: defines records as nodes, and relationships between records as edges
        - e.g. Neptune, OrientDB, neo4j



-
- can be directed or undirected depending on situation and privacy requirements
- graph query languages: Cypher, GQL, Gremlin,

Database languages:

- Data definition language (DDL): create/delete objects
- Data manipulation language (DML): CRUD

- For relational databases, both DDL and DML are SQL

Data manipulation language

- Two types:
  - **Procedural**: user specifies what data are needed and how to get it
  - **Declarative**: user specifies what data, but not how to get it
    - SQL is declarative

- Query: a written expression to retrieve or manipulate data
  - Query language: language it is written in

When SQL isn't enough

- The most common option is to write an ETL job in your favorite language
  - **E**xtract the data from the database using a database connection driver
  - **T**ransform the data using your favorite language
  - **L**oad the data into a new table using the same driver

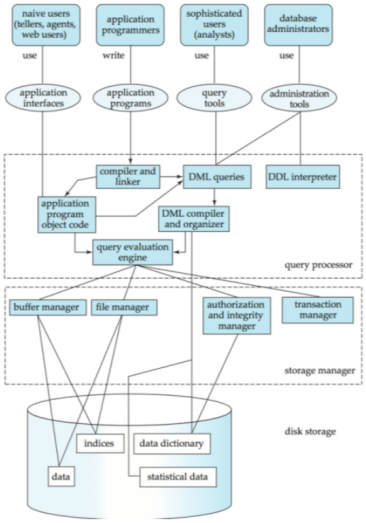
Data definition language:

- Specifies a schema:
  - collection of attribute names and data types
  - consistency constraints

    - optionally, storage structure and access methods
    - Types of constraints a DDL can specify:
        - domain constraints: (e.g. int, enum)
        - referential integrity: no references to non-existent records
        - assertions: business rules
        - authorization: permissions

Data storage and querying:

- Databases have a storage manager that allows us to typically not worry about how data is laid out on disk
- Databases can be on the order of GBs to TBs. Some organizations have even exceeded the PB milestone
- Most of the data cannot fit in RAM, and must be read in from disk
- What data structures allow for fast lookup?
    - Trees, tries, hash tables
- Query manager: When a query is executed, the DML statements are organized into a query plan that consists of low-level instructures that the query evaluation engine understands toperform some operation on the data
    - Important for debugging problems in queries



Relational databases:

- Collection of relations/tables
- Each relation consists of sets of tuples/records, each containing same attributes in same order
- each tuple/record forms a row of a table
- each attribute of a record forms a column in a table
    - each attribute has a unique name and particular datatype
- tables/relations can be related via some attribute

superkey: set of one or more attributes that uniquely identifies a tuple and distinguishes it from all other tuples

video_id	title	channel	cat_id	views
XpVt6Z1Gjjo	1 YEAR OF VLOGGING	Logan Paul Vlogs	24	4394029
cLdxuaxaQwc	My Response	Pewter	22	5845909
Ayb_2qbZHm4	Honest College Tour	CollegeHumor	23	859289
EVp4-qjWVJE	Chargers vs. Broncos	NFL	17	743947
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

What are the superkeys? It depends:

- Instance: For the instance of the four shown rows, all 5 attributes are superkeys
- Context: For the entire table, video-id, and all subsets of R (relation) containing video_id, are possible superkeys

To find superkeys:

- Compute all subsets of attributes
- Eliminate any subsets that fail the superkey definition definition

More about superkeys:

- In every relation R, there exists at least one superkey
- The most basic superkey is the trivial superkey, which is the set of all attributes of R
- In every relation R, there exists at most 2^n-1 superkeys
- Empty set cannot be superkey
- If k is a superkey of R, and s is another attribute, then k U s is a superkey

Candidate key

- A **minimal superkey/candidate key** is a superkey such that no subset of its attributes from a superkey. There may only be one NULL value in a candidate key
- A **primary key** is a single candidate key chosen by the database designer to enforce uniqueness
    - must be unique, must be NOT NULL
    - to choose a primary key consider:
        i. use case
        ii. size

- composite primary keys may not be minimal

	dl_number	license_plate	state	name	auto_make_model
	B9614355	I♡UCLA	CA	Joe Bruin	Ford F-150
	N19154	GENEBLOCK	CA	Gene Block	Tesla X
	8675309	JENNY	CA	Jennifer Lopez	BMW 750i
	B9614355	I♡UCLA	WA	Joe Bruin	Ford F-150
	8675309	JENNY	NY	Jennifer Lopez	Tesla X
	B9914561	GO RAMS	CA	Sean McVay	BMW 750i
	1212121	JENNYFROMBLOCK	CA	Jennifer Lopez	AMC Pacer

Candidate keys: dl_number U state, license_plate U state

Brute force algorithm for finding keys

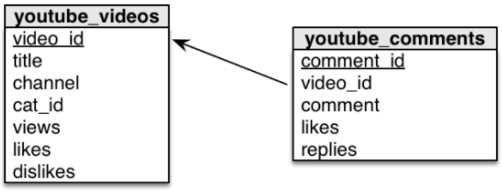
1. Find all superkeys
2. Of the keys remaining, look at the set of attributes in them
    a. look at the att
3. If there is more than one key remaining, pick one of them tobe the primary key based on the use case.

Foreign key: an attribute, or set of attributes that can be used to link together related tuples of two relations in some way. A foreign key reference the primary key of another relation

- often used to impose **referential integrity constraints**
    - protect data in one relation from becoming orphaned or inconsistent due to changes made to data in another relation
- cascade: delete anything that relies on this particular object
- setting foreign key to NULL allows you to bypass constraints
- foreign keys should not be used for joins

Schema diagrams

- Box for each relation
- Relation name on top
- Primary key is first attribute(s) listed, should be underlined
- For foreign keys, point from referring relation to referred relation (from foreign key to corresponding primary key)



-
- can be described as:
    - youtube_videos(video_id, title, channel, cat_id, views, likes dislikes)
    - youtube_comment(comment_id, video_id, comment, likes, replies)
    - **no way to represent foreign keys in this format**

Relational Algebra

- Databases come in many shapes and forms, but all essentially support same set of basic operations (CRUD)
- **relational algebra** is a semantic system for modeling operations on relational data. It provides a theoretical foundation for the relational model as well as query languages such as SQL
- operations we use in the relational model:
    - filtering tuples
    - extracting subsets of attributes from tuples
    - forming all pairs of tuples between two relations
    - merging tuples together based on some attribute
    - performing standard set theory operations on tuples
    - aggregating groups of tuples

        1. Selection σ (or restriction)
        2. Projection Π
        3. Cartesian Product $\times$
        4. Natural Join $\bowtie$ and Theta Join
        5. Aggregation γ ($\mathcal{G}$ is also used for the same)
        6. Rename ρ
        7. Set Union $\cup$
        8. Set Difference $-$ and Set Intersection $\cap$

    -

- **selection**
    - retrieves a subset of tuples from a single relation that satisfies a particular constraint and returns a new relation which is a subset of the original

$$\sigma_{\psi}(R) = \{t \in R : \psi(t)\}$$

- ψ : filter expression

- $\sigma_{\text{likes} > \text{dislikes}}(\texttt{youtube\_video})$: all video tuples where # likes > # dislikes

  We can also build up more complex predicates using conjunction $\wedge$ (and) or disjunction $\vee$ (or).

  Logical operators we can use in predicates include $=$, $\neq$, $<$, $>$, $\leqslant$, $\geqslant$, $\neg$ (not).

- example:

```
SELECT *              //get all attributes
FROM youtube_video
WHERE likes > dislikes  //filter expression
AND views > 1000000  //filter expression
AND cat_id = 24 //filter expression
```

relational algebra: $\sigma_{l > d \wedge views > 1000000 \wedge cat\_id = 24}$ (youtube_video)

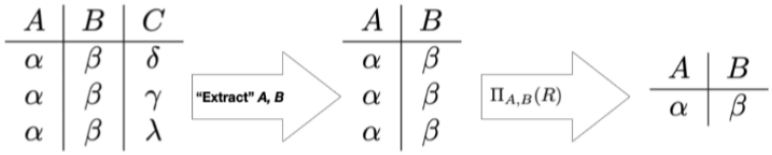
* : return all columns of a particular table
σ != SQL SELECT. It is instead analogous to **WHERE** clause (or **SELECT * WHERE**)

- **projection**
  - simply extracts attributes from a set of tuples, removes duplicate tuples

    $$\Pi_{a_1,\ldots a_n}(R) = \{t[a_1, \ldots a_n] : t \in R\}$$

  - projection is typically the last operation done on a relation
  - can be *generalized* to create new attributes (constructs read-only (not part of schema) ephemeral attributes)

    1. Apply arbitrary expressions to existing attributes (For example $\Pi_{\text{likes}/(\text{likes}+\text{dislikes})}$)
    2. Apply arbitrary expressions to existing columns to create another one (For example $\Pi_{\text{likes} + \text{dislikes} \rightarrow \text{interactions}}$)
    3. Rename attributes. ($\Pi_{\text{likes} \rightarrow \text{thumbs\_up}}$)



- Π: **SELECT DISTINCT (AS)**

- e.g. write a query to extract the title and channel for all videos that have more than 1 million views and more likes than dislikes
- $\Pi_{\text{title, channel}}(\sigma_{\text{l > d ^ views > 1000000}}(\text{youtube\_video}))$
- *attribute* *tuple filter* *relation*

```
SELECT DISTINCT
    title,          } Π
    channel
FROM
    youtube_video
WHERE
    likes > dislikes
    AND             } σ
    views > 1000000
```

- $\Pi_g\left(\sigma_{f(a_1)\wedge\ldots\wedge f(a_n)}(R)\right)$ == $\Pi_g\left(\sigma_{f(a_1)}\left(\ldots\sigma_{f(a_n)}(R)\right)\right)$

- **cartesian product**
  - combines tuples from two relations, in all possible combinations of tuples

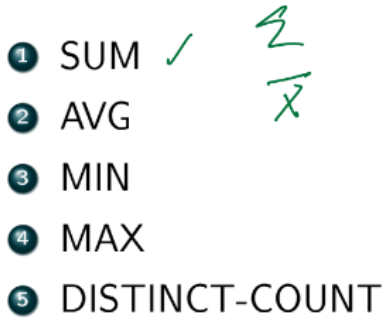
    $$R \times S = \{r \cup s | r \in R \wedge s \in S\}$$

    - all pairs of tuples from R and S concatenated
  - slow to compute, require a lot of RAM, a lot of disk space
  - Cartesian product used to motivate the discussion of a **join**
  - **join**
    - when we want to merge tuples from two different relations R and S based on some contextually related attribute(s) in both relations
    - the result is another relation where each tuple contains data from the tuples of both R and S
    - the attribute(s) used for the join is/are usually referred to as the **join key**
      - join key does not need to be unique

- good database design relies on separation of concerns to reduce redundancy
    - normalization is used to do this
- joined relations are denormalized. We do not want to store data in this format
- **natural join** ⋈
    - RDBMS automatically picks attribute(s) k to join on – the attribute names common to both relations
    - the most simple natural join is defined using Cartesian product
        - $$R \bowtie S = \Pi_{R \cup S} (\sigma_{R.k=S.k} (R \times S)) = \{r \cup s | r \in R \wedge s \in S \wedge (r[k] = s[k])\}$$
    - edge case:
        - if R and S have no common attributes, then:
            - R ⋈ S = R x S
        - if R and S have common attributes but no matches on those attributes, then:
            - R ⋈ S = { }
- **equijoin**: when tuples are combined using equality of values in common attributes (natural join is an equijoin and inner join, except when it is cartesian product/cross join (no common attributes))
- **theta join:** θ is a constraint that specifies how tuples from both relations will be joined
    - $$R_1 \bowtie_\theta R_2 = \sigma_\theta (R_1 \times R_2)$$
    - e.g. R $\bowtie_{R.A=S.A}$ S
    - very powerful compared to natural join
    - if θ consists entirely of equality constraints, it is an equijoin, if not, it is a non-equijoin
- every join is either a natural join or a theta join
- every join is either an inner join or an outer join
- **inner join**
    - merges two tuples together if and only if the join condition is satisfied. The merged tuple is then part of the join result
    - L $\bowtie_{L.A=R.A}$ R
- **outer join**
    - merges two tuples together even if the join condition is not satisfied, but in a specific manner
    - **left outer join**
        - we keep the tuple from L. When we look at a tuple from R and the join condition matches, we merge the tuples together just like with the inner join
            - for tuples in L with no corresponding tuples in R, we fill in attributes of R with NULL
        - L $⟕_{L.A=R.A}$ R

- **right outer join**
- **full outer join**

- **aggregation**
  - apply an aggregation function to groups of tuples in a relation to summarize them.
    - common aggregation functions:



    -
  - $_{\text{major}}\gamma_{\text{AVG(gpa)}}$(ugrad_students): average GPA by major
  - global aggregate: $\gamma_{\text{AVG(gpa)}}$(ugrad_students): average GPA for all records
  - $\gamma_{\text{AVG(gpa), MIN(sleep)}}$(ugrad_students): average GPA and minimize sleep for all records
- **rename**
  - $\rho_S(R)$ renames relation/expression R to S
  - $\rho_{a/b}(R)$ renames attribute b in relation R to a
  - R x R is invalid, so we can do $\rho_L(R)$ x R

- **Set Theory:**
  - when performing an operation on two sets in relational algebra, both sets must be compatible (same # of attributes, same attribute names, same domain/type for each attribute)
  - U: **union**
    - all records that are in R, S, and both R and S
  - – : **set difference**
    - R – S: retrieves tuples in R but not in S
  - ∩: **set intersection**
    - not a join: joins should join incompatible relations

$$R \cap S = R - (R - S)$$

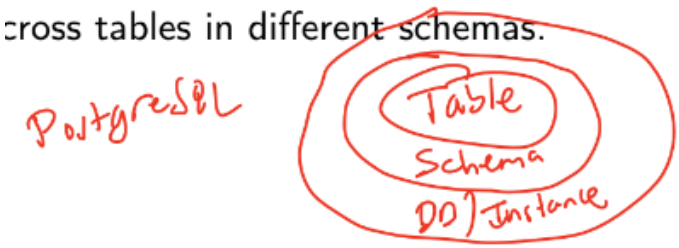
    -

1. σ, Π, ρ have highest precedence
2. $\times$ and $\bowtie$
3. $\cap$
4. $\cup$ and $-$

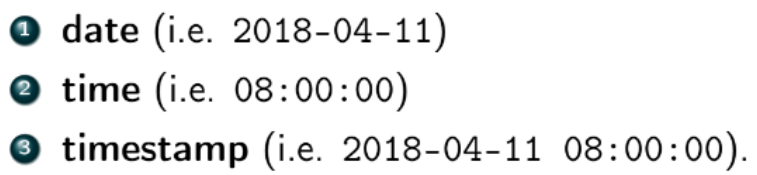
These are properties that we did not already cover, if you're interested:

1. $\sigma_A\sigma_B(R) = \sigma_B\sigma_A(R)$ (idempotent and commutative)
2. $\sigma_A(R - S) = \sigma_A(R) - \sigma_A(S) = \sigma_A(R) - S$
3. $\sigma_A(R \cap S) = \sigma_A(R) \cap \sigma_A(S) = \sigma_A(R) \cap S = R \cap \sigma_A(P)$
4. $\Pi_{a_1 \ldots a_n}(\sigma_A(R)) = \sigma_A(\Pi_{a_1 \ldots a_n}(R))$ were fields in $A \subset \{a_1, \ldots a_n\}$
5. $\Pi_{a_1 \ldots a_n}(R \cup S) = \Pi_{a_1 \ldots a_n}(R) \cup \Pi_{a_1 \ldots a_n}(S)$ (distributive over set **union**)
6. $(R \times S) \cup (R \times T) = R \times (S \cup T)$

PostgreSQL Intro



-
- ANSI SQL compliant
  - is open-source
  - has query execution plans are readable by the user.
  - has a real security and authorization model.
  - has an extensible and flexible type system similar to programming language.
  - is ACID compliant, has more modes of replication etc.
-
- ANSI SQL types:
  - numeric
    - int(eger)
    - smallint (restricted domain)

        - numerical(p, d): fixed-point # containing p digits, d of which appear after the decimal sign
        - real, double precision: floats and doubles
        - float(n): float with precision of at least n digits
    - string/text
        - char(n): fixed-length character string of length n
        - varchar(n): variable-length character string with a max of n characters
    - binary
    - dates and times
        - 

    - null: unspecified or missing value (empty/blank space in PostgreSQL)
- PostgreSQL types
Type	Bytes	Range
smallint	2	-32768 to +32767
integer	4	-2147483648 to +2147483647
bigint	8	-9223372036854775808 to +9223372036854775807
    - exact integer types:
        - smallint: 2 bytes
        - integer: 4 bytes
        - bigint: 8 bytes
    - arbitrary precision fixed-point types:
        - decimal(p, d): same as numeric
            - used for currency, GPA
        - NaN (not a number): must be specified with quotes when referred to explicitly
            - NaN == NaN
            - NaN > everything else except infinity
            - operations on NaN yield NaN
    - floating point types
        - real (1e-37 to 1e37, precision of 6 decimal digits)
        - double precision (1e-307 to 1e308, precision of >= 15 digits)
        - ‘Infinity’, ‘-Infinity’, ‘NaN’
    - char(n), varchar(n), text (1 GB)
        - n = # of characters, *not* # of bytes
    - bytea (hex of escape format)
    - boolean

TRUE	FALSE
't'	'f'
'true'	'false'
'y'	'n'
'yes'	'no'
'on'	'off'
'1'	'0'

- bit(n), varying(n): bitstrings
- timestamp (w/o timezone): 8B
- timestamptz (w/ timezone): 8B
- date: 4B
  - yyyy-mm-dd
- time: 8B
  - hh:mm:ss. …
- timetz: 12B
- interval: 16B
- ENUM: string represented internally as an integer
  - CREATE TYPE mood AS ENUM (‘sad’, ‘neutral’, ‘happy’, ‘sad’)
  - CREATE TABLE person (
    - name varchar(255),
    - current_mood mood
  -);
- spacial and geometric data types
  1. determine if a set of points is within some polygon on a map (i.e. geofencing, reverse 911)
  2. determine where two polygons overlap/intersect (i.e. triangulation?)
  3. determine which points lie along a line (i.e. determine which street a car is currently on given a point)
  4. finding points within a radius (i.e. Indian restaurants within 5 miles of me)

Name	Storage Size	Description	Representation
`point`	16 bytes	Point on a plane	(x,y)
`line`	32 bytes	Infinite line	{A,B,C}
`lseg`	32 bytes	Finite line segment	((x1,y1),(x2,y2))
`box`	32 bytes	Rectangular box	((x1,y1),(x2,y2))
`path`	16+16n bytes	Closed path (similar to polygon)	((x1,y1),...)
`path`	16+16n bytes	Open path	[(x1,y1),...]
`polygon`	40+16n bytes	Polygon (similar to closed path)	((x1,y1),...)
`circle`	24 bytes	Circle	<(x,y),r> (center point and radius)

- JSON
- XML
  1. Network Address Types (`cidr`, `inet`, `macaddr`)
  2. UUID
  3. Arrays (likely used sparingly as it clashes with relational model)
  4. Composite types (similar to `struct`)
  5. Ranges of primitive types, from/to.
  6. Full text search (bag-of-words)
-

Data Management

- datatype examples
  - upload date: timestamptz
  - video length/duration: interval
  - description of video: text
  - flag indicating if video is public, private, united: enum
  - video content: bytea
  - user location: point

```
CREATE TYPE privacy_setting AS ENUM ('public', 'private', 'unlisted');
CREATE TABLE youtube_video (
        video_id        character(11),
        title           varchar(100) NOT NULL,
        channel         character(24) NOT NULL,
        cat_id          smallint,
            -- could also be ENUM with names
        likes           integer,
            -- can always promote
            -- if every human likes this video, int isn't big enough
        dislikes        integer,
        views           integer,
        post_date       timestamptz NOT NULL DEFAULT CURRENT_TIMESTAMP,
        duration        interval,
            -- '2m 5s'
        description     text,
        privacy         privacy_setting DEFAULT 'public',
        location        point,
        content         bytea NOT NULL,
        extra_data      jsonb,
        PRIMARY KEY (video_id),
        FOREIGN KEY (channel) REFERENCES youtube_channel(channel_id)
);
```

  -
    - FOREIGN KEY (channel) REFERENCES youtube_channel(channel_id) ON DELETE CASCADE *: if channel is deleted, all videos in channel are also deleted

PostgreSQL

- in PostgreSQL, a schema is a collection of tables/relations that all have similar use case.
  - Creating a relation/table
    - CREATE TABLE IF NOT EXISTS
  - Changing a table's schema
    - avoid changing schemas if you can
    - promote, don't demote

- don’t delete columns, abandon them
- don’t add columns
- if you need more columns:
    - have a JSON blob column
    - create second table with join key
- otherwise:
    - ALTER TABLE
        - add and drop attributes
        - add and drop primary and foreign keys
        - rename relations and attributes
        - add indices and constraints
        - add or drop partitions
        - change data types of columns

```
ALTER TABLE youtube_video
  DROP cat_id,
  ALTER COLUMN video_id TYPE char(12) USING UPPER(video_id),
    -- add a 12th character to video_id and convert all video_id to uppercase.
  ADD flagged bit,
    -- add a boolean flag for flagged videos.
  ADD rowno int,
  DROP CONSTRAINT youtube_video_pkey
    -- In Postgres we drop the PK constraint
    -- The constraint looks like relationname_pkey
    -- Does not delete the column
    ADD PRIMARY KEY video_id;

ALTER TABLE youtube_video RENAME COLUMN video_id TO id;
ALTER TABLE youtube_video RENAME TO youtube_videos;
    -- finally, rename the table.
```

- Dropping, deleting, truncating
    - DROP TABLE/SCHEMA/DATABASE or DROP TABLE/SCHEMA/DATABASE IF EXISTS
    - TRUNCATE: removes all records from table
    - DELETE FROM … WHERE … : delete particular rows

- Extracting data from table
    - SELECT

```
SELECT
        col1,
        col2 AS new_col2
        ...
        cold
FROM relation_name
WHERE conditiontrue;
```

-
- $\Pi_{cols1,\ col2->new\_col2,\ \ldots\ cold}(\sigma_{conditiontrue}(relationname))$
- :: type cast

```
SELECT
        video_id,
        title,
        (views / 1000000)::float AS millions_views,
        (likes / (likes + dislikes))::float AS pct_liked
FROM youtube_video;
```

-

- WHERE clause
  - WHERE acts on values in columns, one-to-one transformation functions
    - *no aggregation functions*
    - *cannot use AS alias in WHERE clause*
  - =, >, <, >=, <=, !=
  - views “BETWEEN 1 AND 3” is the same as “views <= 3 AND views >= 1”
- ordering rows
  - ORDER BY [column(s)]
    - ***can** use AS alias in ORDER BY clause (only clause you can use aliases)*
    - default: sort in ascending order
      - ORDER BY … DESC: change to descending order

```
SELECT
    uid, last, first, gpa
FROM bruinbase
ORDER BY gpa DESC, last, first
LIMIT 3;
-- limits the output
```

      -
      - NULLS FIRST, NULLS LAST to specify if NULLS come first or last (default–they come first)
    - *use column names*
- aggregation
  - aggregations over entire relation
  - aggregations by group

- GROUP BY

```
SELECT
        major,
        AVG(gpa) AS average
FROM bruinbase
GROUP BY major;
```

- As soon as we use an aggregation, whether it's an aggregation function or a `GROUP BY` clause, **every** column in the `SELECT` must either be:
  1. involved an aggregation function; or
  2. be part of the `GROUP BY`
- filtering on aggregations
  - HAVING

```
SELECT
        major,
        AVG(gpa)::decimal(3,2)
FROM bruinbase
GROUP BY major
HAVING AVG(gpa) < 3.95;
```

Suppose I want to do the following analysis:

1. I want to use only undergraduate GPAs in this calculation.
2. I want the average GPA of each major.
3. I want to look at only majors that have an average GPA below 3.95
4. Of those, I want to order them in descending order by average.
5. Display the top 2.

- WHERE, GROUP BY/AVG, HAVING, ORDER BY DESC, LIMIT

```sql
SELECT
    major,
    AVG(gpa)::decimal(3,2) AS average
FROM bruinbase
WHERE career = 'UG'
GROUP BY major
HAVING AVG(gpa) < 3.95
ORDER BY average DESC
LIMIT 2;
```

- When writing a query, use the following order. You will get an error if you don't.

  SELECT
  FROM
  JOINs
  WHERE
  GROUP BY
  HAVING
  ORDER BY
  LIMIT
  OFFSET

- The order in which the statements are executed is very different.

  FROM
  ON
  JOIN
  WHERE
  GROUP BY
  HAVING
  SELECT
  DISTINCT
  ORDER BY

SQL Joins

- joins are stored in a buffer
- natural join is an SQL anti-pattern (it is dangerously implicit)
    - **inner join:**

```
SELECT
  instructor_name,
  course_name
FROM instructor
INNER JOIN course
ON instructor.ID = course.ID
   -- INNER is default so, optional
INNER JOIN course_offering
ON course.course = course_offering.course;
```

    -
    - or

```
SELECT
  instructor_name,
  course_name
FROM instructor r
INNER JOIN course s
ON r.ID = s.ID
INNER JOIN course_offering t
ON s.course = t.course;
```

    -
- self joins can be used for network problems, particularly DAGs and trees

the SQL query for this self join for FOAF look like? *Hint: There is no special syntax for a self join.*

```
SELECT DISTINCT
  L.user_id AS user,
  R.friend_user AS foaf
FROM friends L
JOIN friends R
ON L.friend_user_id = R.user_id AND
   L.user_id != R.friend_user_id;
```

SELF JOIN

    -
- **outer join**
    - benefit of splitting relation into two tables:
        - only need to append when updating records, instead of searching (e.g. if end_time is nonexistent, we would have to search for the record before updating it
    - in SQL, you need to specify LEFT, RIGHT, or FULL before JOIN
        - JOIN on its own implies inner join
        - OUTER keyword is optional
        - FULL JOIN

        - combines results from LEFT JOIN and RIGHT JOIN
- non-equi self join
    - when we want to compare rows to each other given a particular ordering
    - using it as a sliding window
    - Write a query that computes, for each transaction, the number of cumulative chargebacks each customer had incurred in the previous 5 days and order by transaction date, most recent at top.

```
SELECT
    L.trans_id,
    L.customer_id,
    SUM(R.result) AS chargebacks
FROM purchase L
JOIN purchase R
ON L.customer_id = R.customer_id AND
    L.transtime - R.transtime + 1 <= 5 AND
    R.transtime <= L.transtime    -- sneaky condition
GROUP BY L.trans_id, L.customer_id
ORDER BY trans_id DESC;
```

- **cross join** between R and S: cartesian product of R and S

Nested queries

- innermost subqueries are always executed first
- in PostgreSQL, a subquery can be nestd inside SELECT, INSERT, UPDATE, DELETE
- nested queries are used for:
    - constructing derived tables to chain together processing or aggregation steps in the FROM clause

```
SELECT
        uid, last, first, mi, scores.career, midterm,
        (midterm - mean) / sd AS z_score
FROM (
        SELECT
                career,
                AVG(midterm) AS mean,
                STDDEV(midterm) AS sd
        FROM midterm_scores
        GROUP BY career
) aggregated
JOIN midterm_scores scores
ON scores.career = aggregated.career;
```

        - derived table must have alias

    - computing constants for use in filters using scalar subqueries (exactly one value)

```
SELECT
        uid, last, first, mi, midterm
FROM midterm_scores
WHERE midterm > (
        -- this subquery makes the constant
        -- available to the query.
        SELECT
                AVG(midterm) + 0.5 * STDDEV(midterm)
        FROM midterm_scores
);
```

Compute $\mu + \frac{1}{2}\sigma$

- It is important to realize that the following **does. not. work.**:

```
SELECT
    uid, last, first, mi, midterm
FROM midterm_scores
WHERE midterm > AVG(midterm) +
       0.5 * STDDEV(midterm);
         -- Aggregate functions are not allowed in WHERE
```

because AVG and STDDEV are aggregation functions and must be computed first in order to use them.

    - semijoin/antijoin
    - a *correlated subquery* is a subquery that uses names/aliases external from the subquery. *<u>These are very inefficient</u>*. The subquery is fully executed for every single row
    - subqueries vs joins:
        - joins can be expensive on larger tables, so we should filter before we join

Adding new rows

- INSERT INTO relation VALUES (‘val1’, … , ‘valn’) or
- INSERT INTO relation (col1name, …, colnname) VALUES (‘val11’, … ‘val1n’), (‘valn1’, ‘valnn’)

Modifying rows

- UPDATE relation SET column = new_value WHERE condition

Data integrity constraints

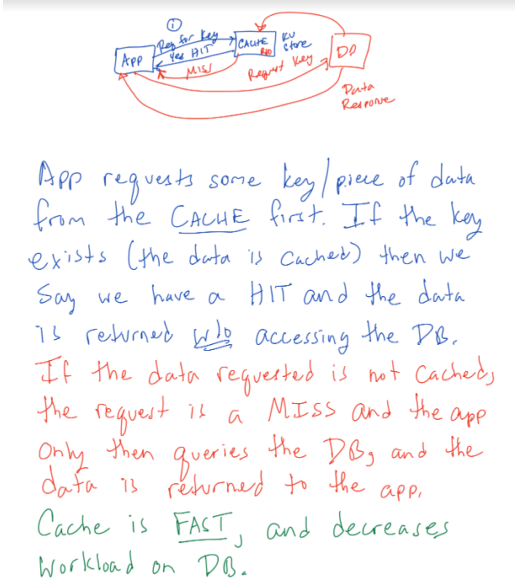
- CHECK
    - CONSTRAINT constraint_name CHECK (expression)
    - clause can be a boolean expression or subquery
    - can only be used per row
- pros and cons of enforcing constraints in database

    - + enforces data integrity
    - + sets syntax regardless of the app language
    - + don't need to trust app developer
    - + changes to app don't break this logic
    - - limited in functionality
    - - less flexible as app changes
    - - more CPU load used on checking constraints
- pros and cons of enforcing constraints in application
    - + constraints can be more complex, closer to the developer. Complex data structures
    - + failures easier to debug
    - - bad user input
    - - need to reimplement if stack changes
    - - not as fast
- best practice is to not delete or update values of primary keys if this is our only source of the data
    - just stop using it
    - RESTRICT (disallow deletion), CASCADE, SET NULL (set value of foreign key to NULL when it is deleted or updated in the referenced table (ON action SET NULL), rarely useful) can be used to update value of primary key
        - by default RDBMS will give an error (RESTRICT by default)
    - ON DELETE/UPDATE SET NULL/RESTRICT/CASCADE
- UNIQUE: forcing a (set of) columns that is not a primary key to be UNIQUE
    - can enforce uniqueness on an individual column, composite of columns (UNIQUE(col1, col2)), or multiple columns independently (UNIQUE col1, UNIQUE col2)

Two philosophies for development with databases

- pros/cons of performing data operations in backend of web app
    - can use procedural language to write logic
    - easier to move data to another database
    - engineers will be more familiar with language for app rather than database
    - don't need to translate errors from DB to user
    - software/infrastructure changes require rewriting all constraints
    - bad input can break things
- pros/cons of performing data operations in backend of database
    - logic is closer to purpose of database
    - set syntax for constraints, as long as we don't change databases
    - limited in functionality
    - increased CPU usage
- validating constraints:
    - we need JOINs to validate constraints. On every request to add a record, we have to join tables together

    - alternatively, we can pull raw data for all records, ingest it into app and let the app handle the “join”
        - data will need to be synced frequently
- **rule of thumb**
    - all business logic should be encoded in the app
    - all data integrity mechanisms should be left to the database
    - better solution:
        - use a **cache**: precompute joins for a subset of students, and ingest it into the app which can use its own data structures and algorithms to do processing
            - caching involves storing results of past lookups in system that allows very fast retrieval
            - cache can be refreshed every day/hour
            - could also use a **materialized view**



- accessing databases from applications
    - opening a connection to the database using a driver
    - constructing a query
        - various way to construct a SQL query
            - string arithmetic
                - “SELECT * FROM table WHERE column = “ + value + “;”
            - string interpolation
                - “SELCT * FROM mytable WHERE column = {}”.format(variable)
            - prepared query
    - executing the query with a cursor (object that maintains state)
    - iterating over the result set using a cursor
    - closing the ocnnection
- **SQL injection**

    - enter user input such that the input completes an (or multiple) SQL query(ies).
        - `1'; DROP DATABASE students;`
        - `'joebruin' OR 1=1; --`
    - **prepared statements** can protect against SQL injection
- cursors and state
    - cursors keep the database connection open, which wastes server resources
    - cururs ors keep the database in transaction, wasting resources
- logging
    - it's important to log as much as you can
        - login, logout, auth
        - impressions, interactions, conversions
        - visit path through app or site
        - time spent on pages or screens
        - purchases
        - JS and AJAX actions
- privacy
    - minimize copies of private information in logs
    - use join key everywhere else
    - hash super private data
        - hashes need salt (a random string affixed to data before it is hashed) an pepper (similar to salt, but stored in separate table)
    - have retention policies (don't keep data longer than needed)

```
import psycopg2

connection = psycopg2.connect(user="some_user", password="the_p@55w0rd",
    host="localhost", port="5432", database="my_database")
connection.autocommit = True
with connection.cursor() as cur:
    cur.execute("""
    SELECT uid, last, first, mi, gpa, major FROM bruinbase
    WHERE major=%(major_input)s;
    """, {
            'major_input': request_args['major']
        })
    rows = cur.fetchall()
    for result in rows:
        uid, last, first, mi, gpa, major = result
        print("Name: {} {}, GPA: {}".format(first, last, gpa))
connection.close()
# Note that things can and will go wrong, so exception handling
# with try/except/finally is important.
```

-

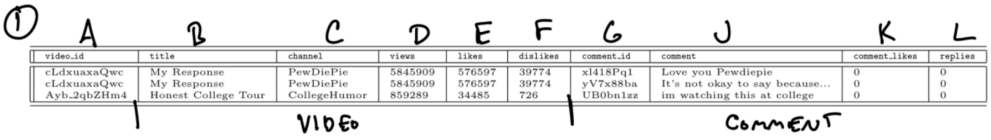
Normalization

- process of refactoring tables to reduce redundancy in a relation.
- reduces redundancy in tables
- separates concerns in database
- prevents data integrity issues
- prevents delays in creating new records
- **functional dependencies**
    - Heath's theorem: if a relation R over attribute set U satisfies functional dependency X -> Y, then R can be split into two smaller relations
    - if a table contains redundant data and is nor normalized

      Given a relation R with sets of attributes X and Y, $X \rightarrow Y$ is true if and only if each value of X is associated with **exactly one** value of Y.

      That is, for all pairs of tuples t_1 and t_2 in R, if $X \rightarrow Y$. then it must be true that if $t_1[X] = t_2[X]$ then $t_1[Y] = t_2[Y]$.

      $X \rightarrow Y$ can be said as X **functionally determines** Y.
    -
    - when a functional dependency X -> Y exists, it means we can split a table into two, one for the functional dependency X -> Y and one for X and everything else in the relation
    - X -> Y = X functionally determines Y
    - finding functional dependencies
        - depends on context, which will be given
        - e.g.

        -
            - A -> BCDEF
            - G -> JKL
            - G -> A
        - to normalize:
            - R_1(A, B, C, D, E, F)
            - R_2(G, J, K, L)
            - R_3(A, G)
    - **properties of functional dependencies (Armstrong's axioms)**
        - Reflexivity: AB -> A
        - Augmentation: if A -> B, then AC -> BC
        - Transitivity: if A -> B and B -> C, then A -> C
    - **other properties of functional dependencies**
        - Composition: if A -> B and C -> D, then AC -> BD
        - Decomposition: if A -> BC, then A -> B and B -> C

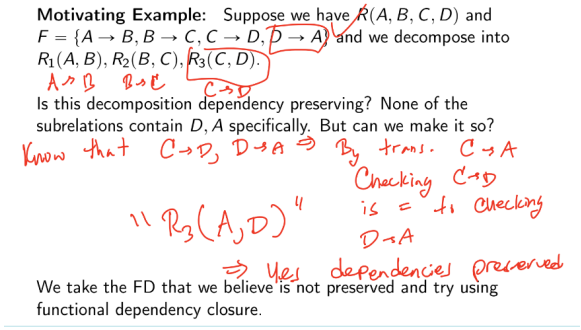
        - Pseudotransitivity: if A -> B and CB -> D, then CA -> D
    - **canonical cover (minimal set)**
        - create singleton RHS (all functional dependencies where right side only has one attribute)
        - remove any extraneous attributes (where LHS has multiple attributes)
            - e.g. if we know A -> C, then AB -> C is extraneous
        - remove duplicate/trivial functional dependencies
        - remove inferred/redundant functional dependencies (we can remove A -> C if we know A -> B and B -> C)
        - apply union rule (A -> B and A -> C means A -> BC)
- denormalized tables
    - issues with INSERT
        - e.g. joining video and comments tables yields denormalized tables; when a new video is created, comments attribute must be specified as null, or video cannot be added to table before comment is posted
    - issues with UPDATE: updating a video requires us to update multiple rows
    - issues with DELETE: deleting a video requires us to delete multiple rows
- why normalization is needed
    1. reduces redundancy in tables
    2. separates concerns in the database.
    3. prevents data integrity issues on insert and delete due to redundancies.
    4. reduces copies of data (more copies = more data likely to become bad)
    -
- NoSQL databases typically do not use normalization

Normal Forms

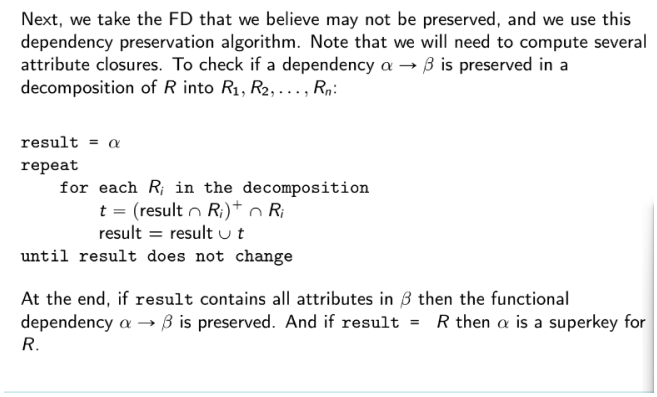
- property of all normal forms up until 3NF and BCNF: **lossless decomposition**
    - all attributes are preserved in decomposition
    - there exists a common attribute between original and decomposed relation
    - common attribute is an superkey for original and decomposed relation
    - to test for losslessness:

1. We can test $\Pi_{\text{Attributes} \in R_1}(R) \bowtie \Pi_{\text{Attributes} \in R_2}(R) = R$
2. We can also test other aspects of the decomposition:
   1. $R_1 \cup R_2 = R$ [all attributes of R present]
   2. $R_1 \cap R_2 \neq \{\}$ [there is a common attribute]
   3. $R_1 \cap R_2$ is a superkey for either R_1, R_2 or both.

- **dependency preservation**
  - the most obvious way to see if a dependency is preserved is to see if both the left and right side are contained in a single subrelation
    - if not, we can move on to other tests:
      - **functional dependency closure**: using the attributes in the subrelations and axioms on F, can we derive a subrelation where the FD in question holds explicitly?



      - **attribute closure**: using all attributes inferred by F, can we rederive all of the attributes in R subject to the decomposition



- **First normal form (1NF)**
  - attributes are flat and simple, no nesting, no collections
  - no repeated groups (that impose a non-sense ordering)
  - there is a unique key
    - (all non-key attributes depend on a key)
  - no null values
- **Second normal form (2NF)**

    - any relation that is 1NF and that does not contain any composite keys
    - non-prime (non-key) attributes must depend on the entire key somehow
    - (each attribute depends on the key, the whole key)
    - either of the following has to be true for every attribute:
        - the attribute appears in a candidate key
        - the attribute entirely depends on an entire key (*not partially dependent* on any composite candidate key)
- **Third Normal form (3NF)**
    - any relation that is 2NF and where all attributes that are not part of any candidate key (non-prime attributes) must depend only, directly, on the candidate key(s) and nothing else (transitivity not allowed)
        - there must exist no transitive way to get to a certain attribute from the candidate key
    - (all non-key attributes depend on a key, the whole key, and nothing but the key)
    - at least one of the following must be true for all functional dependencies:
        - A -> B is a trivial FD
        - A is a superkey for R
        - B is part of a candidate key
- **Boyce-Codd Normal Form (BCNF)**
    - for every functional dependency A -> B, at least one of the following holds:
        - A -> B is trivial
        - A is a superkey for R
    - decomposition to BCNF

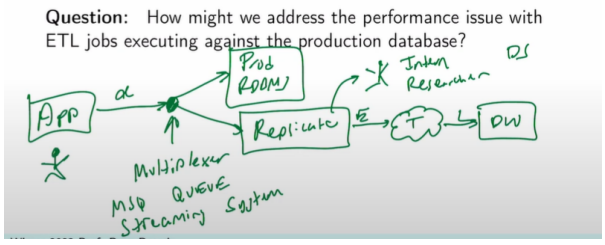
```
for any R in the schema
    if (α → β holds on R and
        α → β is non-trivial and
        α is not a superkey), then
            Decompose R into R1(α ∪ β) and R2(α ∪ γ)
            // α becomes common attributes,
            // γ attributes in R except α, β
Repeat until no more decompositions necessary
```

α → β violate BCNF

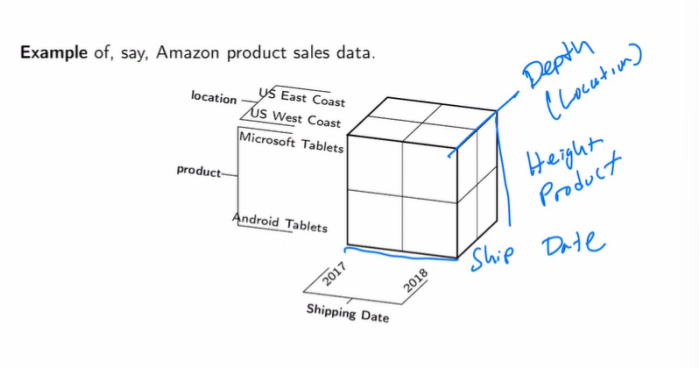
        -
- in read-heavy work, denormalized works better

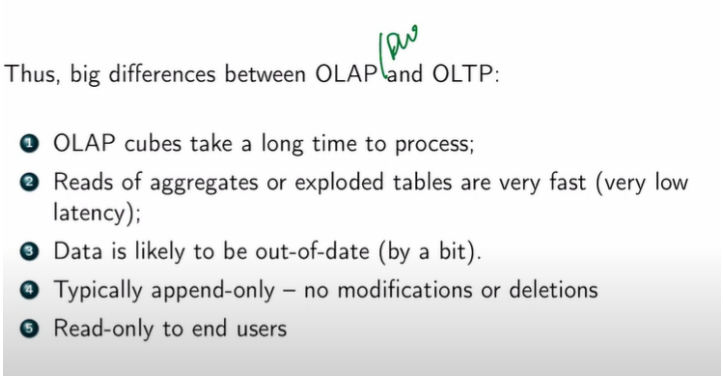
Other Data Management Systems

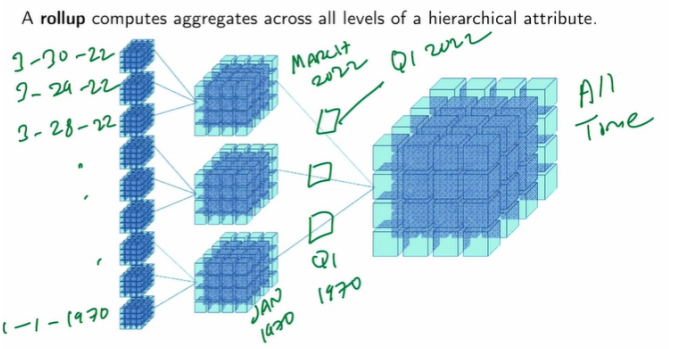
- **OLTP (online transaction processing)**
  - designed for frequent, interactive use
  - random access read/write (transactions)
  - frequently used in production (data read of written by external [unsophisticated] users)
  - simple queries:
    - SELECT FROM
    - JOIN
    - WHERE
    - UPDATE
    - DELETE
    - INSERT
  - abstracts data as a 2D representation called a table
- **OLAP (online analytical processing)**
  - different system, but its abstraction looks similar to OLTP
  - read-only for internal users
    - low-latency reads (SELECT) of aggregated or precomputed data
  - writes can be slow
    - data is loaded into data warehouse during low usage period using ETL (extract, transform, load)
    - 

    - either *all of the data* or *only new data since the last update* is transferred
      - costly
        - solution:
          - run ETL job against replicate
          - dual writes: on each write to production RDMS, write to replicate as well
          - 

    - data that is loaded is often aggregated
      - aggregation is costly, so is usually done overnight and cached
    - data that is loaded is often exploded tables (result of joins)
      - join can be prespecified in schema

- implemented via data warehouses
- data is conceptualized as a **cube** (rows, columns, depth)
  - users still see/work with table
  - 

- used for batch processing
  - reporting
  - dashboarding
- 

- **OLAP operations**
  - **slice**: selects one predominant dimension from cube and returns a new sub-cube
    - WHERE
  - **dice**: selects multiple values from multiple dimensions and returns new sub-cube
    - WHERE + AND
    - typically not used
  - **rollup**
    - computes aggregates across all levels of a hierarchical attribute (e.g. month, quarter, year, etc.)
    - 

```
SELECT
    year,
    month,
    day,
    SUM(sales) AS total_sales
FROM hourly_sales
GROUP BY ROLLUP(year, month, day);
```

year	month	day	total_sales
2020	April	21	200
...	...	...	...
2020	December	31	842
2020	April		2662
...	...	...	...
2020	December		8412
2020			126830
			2526124

- CUBE: all subsets

- **drill down**
  - extracting aggregates at a finer level of granularity
  - **drill up**: the opposite
- **pivot**
  - converts data from *long* format to *wide* format and vice-versa
  - wide:

The first format is **wide**; it involves multiple *columns*, one per homework in this case, and one row per student. For this particular example, the wide format is more natural.

uid	full_name	hw1	hw2	hw3	hw4
246802468	SCHMOE, JOE	100	90	58	
012345678	BRUIN, JOE	85	88	95	63
424242424	BLOCK, GENE	81	70	92	

    -
    - - columns are not fixed (may need ALTER TABLE)
    - - NULL values
    - + less contextual redundancy
    - + easier to understand
    - user-efficient
  - long:

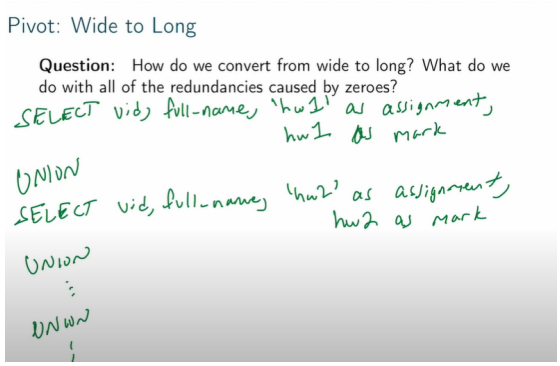
The second format is **long**; it involves multiple rows per student. Each row is a homework assignment. Long data is sometimes called `tall`.

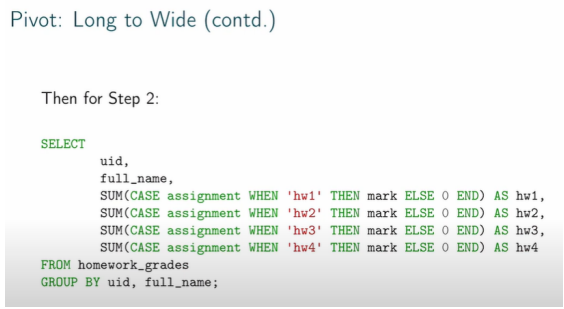
uid	full_name	assignment	mark
246802468	SCHMOE, JOE	hw1	100
246802468	SCHMOE, JOE	hw2	90
246802468	SCHMOE, JOE	hw3	58
012345678	BRUIN, JOE	hw1	85
012345678	BRUIN, JOE	hw2	88
012345678	BRUIN, JOE	hw3	95
012345678	BRUIN, JOE	hw4	63
424242424	BLOCK, GENE D.	hw1	81
424242424	BLOCK, GENE D.	hw2	70
424242424	BLOCK, GENE D.	hw3	92

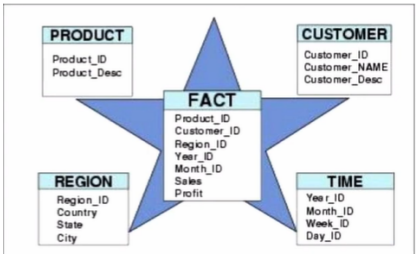
    -
    - can add more HWs without ALTER TABLE
    - no NULL values

        - computer-efficient
    - SELECT CASE
        - switch statement

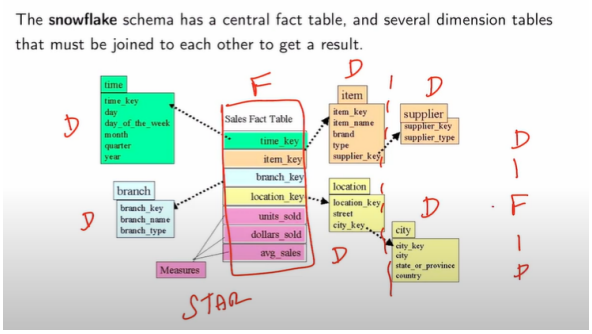
```
SELECT
  ...
  CASE column_to_compare
    WHEN column_is_equal_to_this_1 THEN output_this_1
    WHEN column_is_equal_to_this_2 THEN output_this_2
    ...
    ELSE output_this_if_nothing_matches  --optional
  END AS some_alias
  FROM relation;
```

        -
    - 

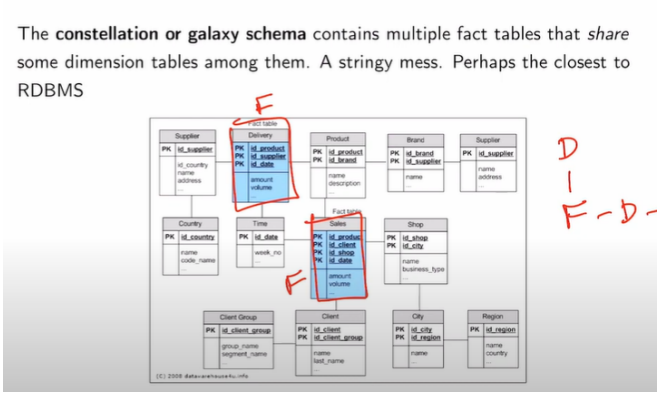
    - 

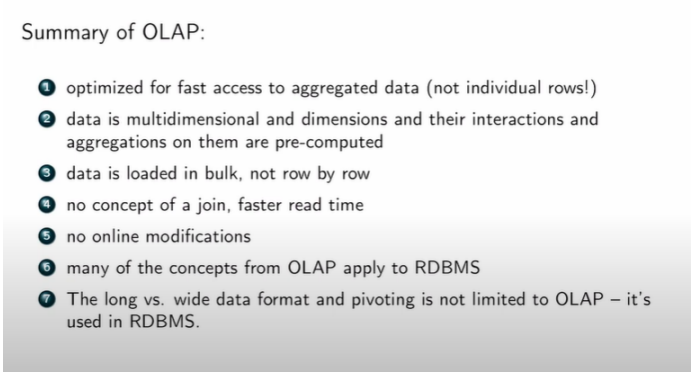
- **schemas**
    - **fact**: raw data + keys
    - **dimension**: metadata: key -> attribute (describes data)
    - **star schema**
        - 

    - **snowflake schema**

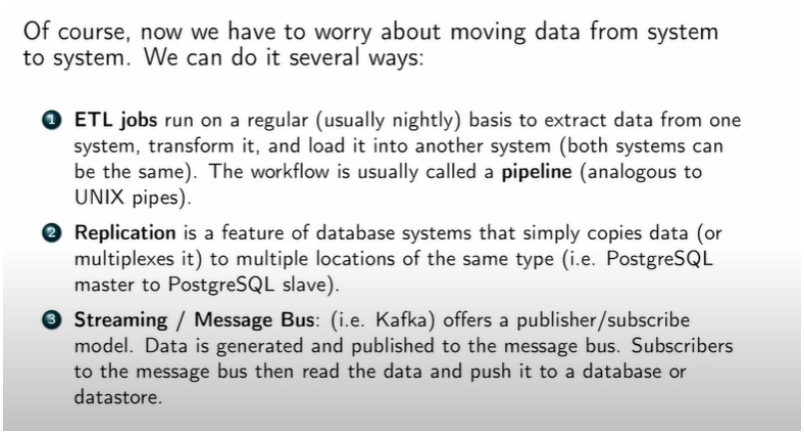
- 

  - **constellation schema**

- 

- 

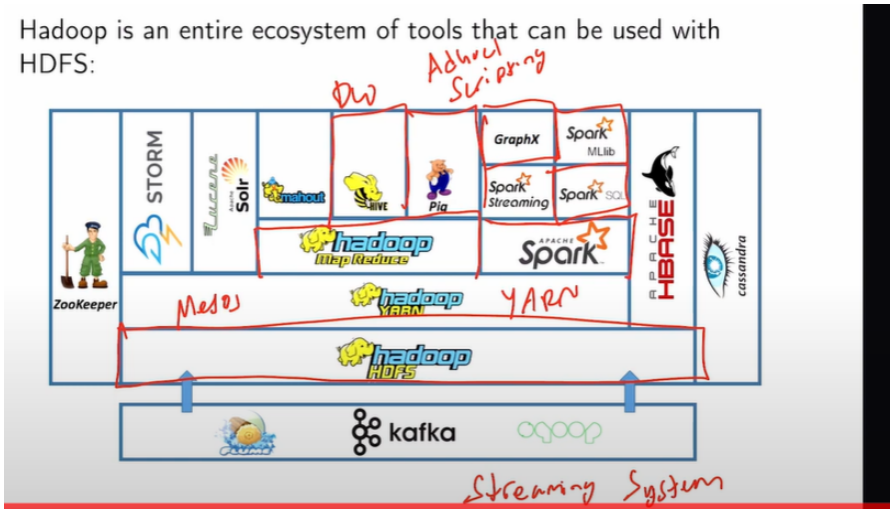
- typically we have a production and development database

- 

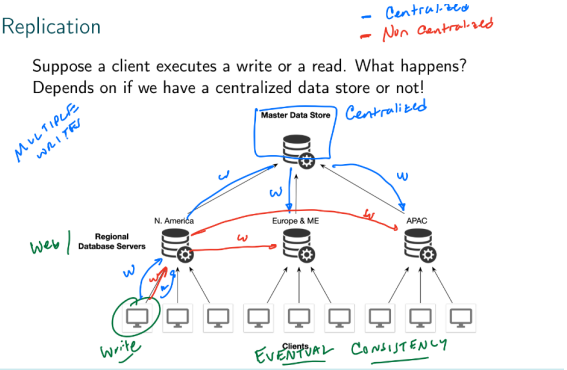
- **data lake**
  - giant repository of raw data
  - different formats in one system
  - HDFS
    - scalable
-

	RDBMS/Database	Data Warehouse	Data Lake
Data	Tables, relational Clean + organized	Redundancy, Cube, Agg.	ANY
Schema	Fixed + Strict	Fixed, Check on insert	None
Price/Performance	OLTP Good R/W Free-ish $$$	Hive, Redshift Mostly $$$	Cheap
Data Quality	Keys + Constraints	Curated Out of date	BAD
Users	App/Personal System	DA, DS, SWE, Sys	Anyone?
Analytics	Not Strong Swift	BATCH Strong Swift	ML?

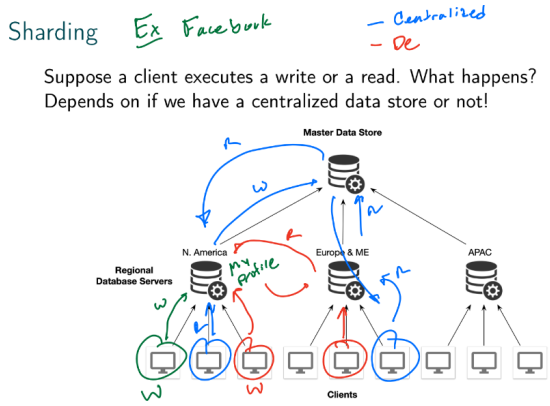
-



- **replication**: all nodes share almost the same data
  - + load balancing
  - + highly available
  - - if we don't sync properly, we lose data
  -



- **sharding**: different nodes contain different data. Shards are usually *geographic*
  - \+ geographic shards minimize latency
  - can be centralized or decentralized (decentralized = data is only stored in local shard, isn't ever moved to master data store)



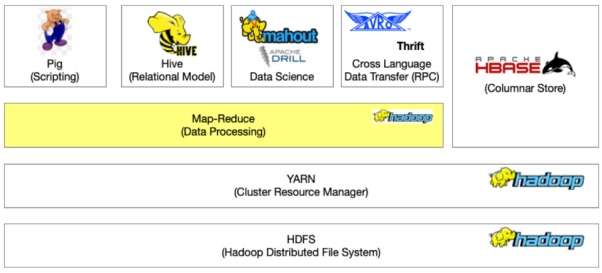
  -
  - centralized system has eventual consistency

Big Data

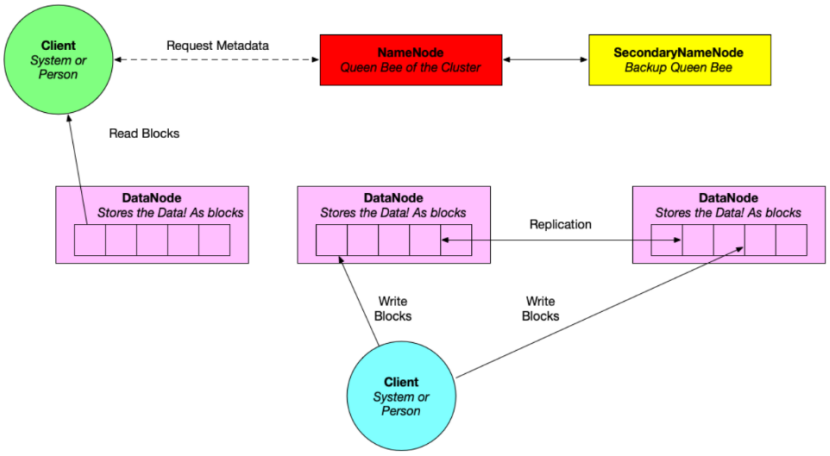
- **big data**:
  - "data that is too large to fit in RAM on one machine"
  - purpose is to maximize usage of compute resources (increase bandwidth)
  - **parallelism**
    - doesn't make data processing faster, but increases throughput allowing us to get a result quicker
    - can make data processing take longer due to overhead

Map-Reduce

- computational method often used to work with big data
- divide and conquer
- two popular systems:
  - **Hadoop**



    -
    - **HDFS**

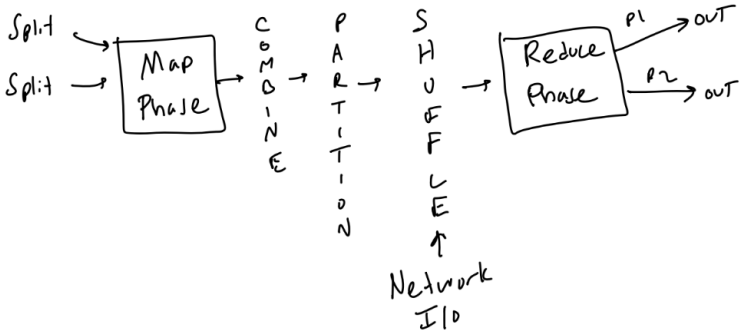


-
- NameNode indexes all blocks

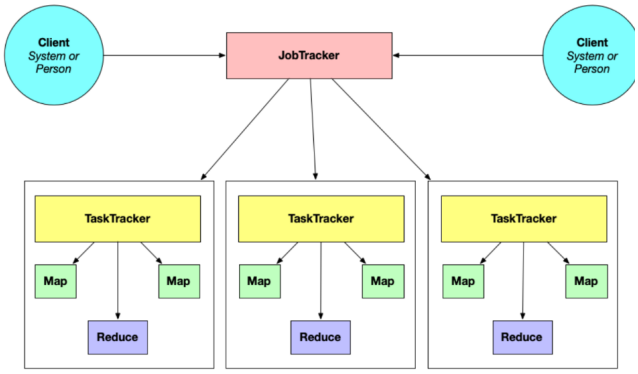
- Spark
- MapReduce stages
  - input
    - InputFormat
      - how records are laid out in files
      - how to divide groups of records into **splits** (contains many records)
        - splits are used to defined parallelism, typically contain 1 block of data
        - too many splits: long time to set up
        - too few splits: reduces parallelism
  - MAP
    - outputs key-value pairs (e.g. word -> (word, count))
  - optional combiner
    - (word, 1), (word, 1) -> (word, 2)
  - partition
    - for every key, we compute a hash, use that hash to assign key-value pair to a certain partition
  - shuffle: merge and sort
    - merge: combine pairs with like keys into (key, [list of values])
    - sort: sort by key
  - REDUCE
    - given (key, [list of values]), we execute a reduce function on the list of values for each key (e.g. if we have a list of word counts, we simply sum them up)
    - problems:
      - key skew
        - certain reducers are going to run for much, much longer than others
        - solutions:

- manually create subkeys
- use a custom partitioner: add salt to key, hashes so that keys are uniformly distributed
    - only need one map+reduce phase

    - output



-
- example of MapReduce that only contains map phase: parser
- we don't partition if we don't reduce



-
- **SQL in MapReduce**
    - WHERE
        - takes place in MAP phase (prefilter on key-value pairs)
    - aggregation
        - GROUP BY: partition
        - aggregation function: reduce
    - JOIN
        - either
    - subqueries encourage parallelism

Spark

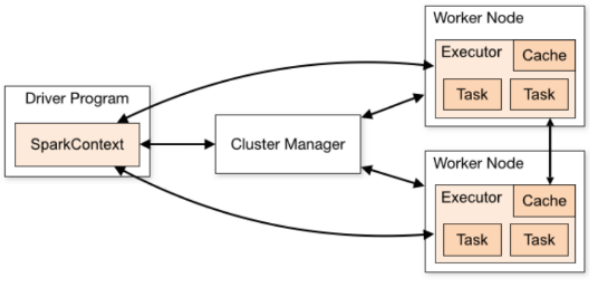
- we want a **pipeline** consisting of a series of algebraic steps
- Apache Spark
    - uses Resilient Distributed Datasets (RDDs)
    - **RDD**

        - can be of any format
        - resilient: when one node fails, there is a replicate somewhere
        - distributed: parts of the data can be on different nodes
        - immutable (faster)
        - useful when data is unstructured
    - computations are stored in RAM, stays there until the user specifically requests to persist them to disk (collect())
    - lazy evaluation: Spark constructs an execution DAG

> With lazy evaluation, we can execute several **transformation** commands and they return immediately, without having to wait. The pipeline will only execute when an **action** is executed. Some actions:
>
> - `head()`, `first()`, `take()`
> - `count()`
> - `collect()`, `collectAsList()` (materialization)
> - `reduce(func)`
> - In DataFrame, `show()`

        - "Only do the work when it's required to show a result."
    - can more easily handle iterative problems
    - other advantages:
        - has native APIs for Scala, Java, Python
        - seamless between Pandas and R
        - can use Hadoop infrastructure if necessary
        - can be run in interactive shell
    - disadvantages:
        - disks are slow
- Joins are the biggest source of poor performance in Spark

> Spark can use HDFS under the hood. A cluster management platform like YARN also abstracts away a lot of the system architecture (compared to Hadoop which uses its own model):



    -

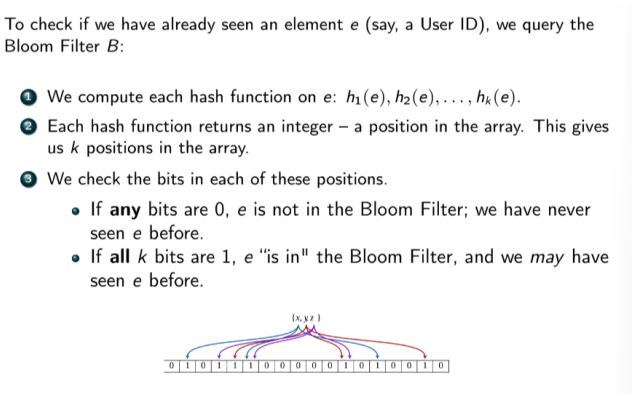
Streaming systems

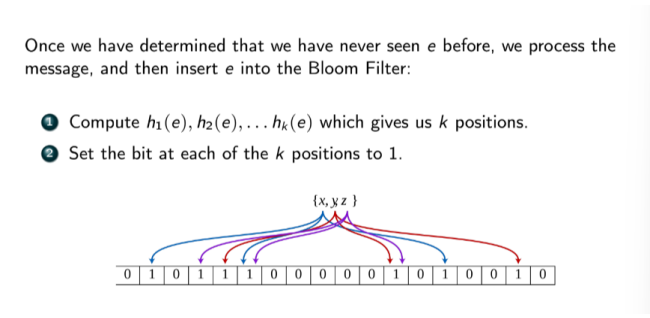
- **Lambda architecture**: an architecture that processes massive quantities of data by using both batch and realtime/stream processing
- **data streams** (data arrives as it is produced, dataset is never complete)
    - RDMS, OLAP, MapReduce, and Spark use bounded/complete data, *not* data streams
    - **simple events**
        - e.g. stock price in last 5 seconds, rate of attacks against server, sentiment during presidential debate
        - we have to do something quickly with data without storing it
    - **complex events**: multiple steps
    - involves individual events or messages, each event identified with a time stamp
    - **producer**/publisher/sender: creates an event or message
        - continuously generate events into RDBMS
    - **consumer**/subscriber/recipient: receives and processes message
        - consumer constantly polls RDBMS for new events
        - polling is expensive. Instead, we want the system to “push” notifications to the consumers
    - problems with producer/consumer:
        - producers are hardcoded to consumers
        - producers are also independent
        - if a consumer goes down,producer needs to manually configure a new consumer
        - decentralized
        - network can be lossy
        - producers can transmit data faster than consumers can process it
    - data loss for data streams:
        - when computing an aggregate:
            - data loss is negligible if data is missing at random
            - receiving duplicate messages is negligible if data is duplicated at random
            - receiving data in wrong order isn’t a big deal
        - when replicating to RDMS:
            - data loss is a problem
            - receiving duplicate messages is negligible; can be prevented with primary key
            - receiving data in wrong order isn’t a big deal

Feature	TCP	UDP
Connection status	A connection (transaction) is required before data transmission begins. Connection is closed after success. Both hosts acknowledge that they are in a connection (`SYN`).	Connectionless. Recipient may or may not know that another machine is trying to send data.
Data sequence	Packets must arrive in order.	Packets can arrive in any order.
Handshaking	Connections initiated with `SYN`, all transmissions must receive an `ACK` or `NACK` response to be complete.	None.
Guaranteed delivery	Atomic. Data is guaranteed to arrive in the correct order, without corruption, or else the sender gets an error. How? Recipient `ACK`s or `NACK`s each packet. Upon `NACK` or timeout, data is retransmitted.	No guarantees. Data can be lost or even be delivered multiple times.
Retransmission of data	Retransmitting lost, or un-ACKed packets is possible.	None.
Flow control	Full flow and congestion control (exponential backoff).	None, usually packet loss in a congested network.
Error checking	Extensive with checksum comparison and `ACK`ing.	Simple, only a checksum comparison.
Transfer method	Byte stream divided into segments.	Entire packages (usergrams).
Broadcasting	Not supported	Supported.
Performance	Slower	Fast
Overhead	Much	Very little

- **message broker**: sits between producers and consumers
  - producers send messages to broker, consumers read message from the broker
  - centralized
  - handles case where producers and consumers come and go
  - when messages are sent to consumer while it's down:
    - broker can buffer the message, and then retransmit to another consumer or spin up a new consumer
    - broker then waits for ACK from consumer. When it receives it, it clears message out of buffer
  - handling multiple consumers:
    - **load balancing**: broker chooses one consumer to receive message either arbitrarily or based on shard or partition key
    - **fan-out**: broker delivers message to ALL consumers in particular groups (similar to multiplexing)
  - **TTL**: valid life period of a message
  - commit ack: sent to producer when message has been successfully processed by consumer
  - if consumer or broker bog down/overflow:
    - we can apply flow control or spin up more consumers
    - we can delete the oldest messages in the queue (if recency is important)
    - we can delete messages in the queue at random (computing aggregates)

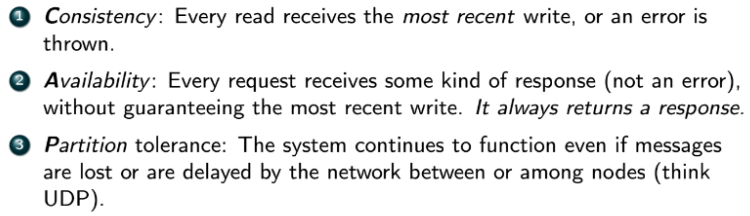
Processing streaming data

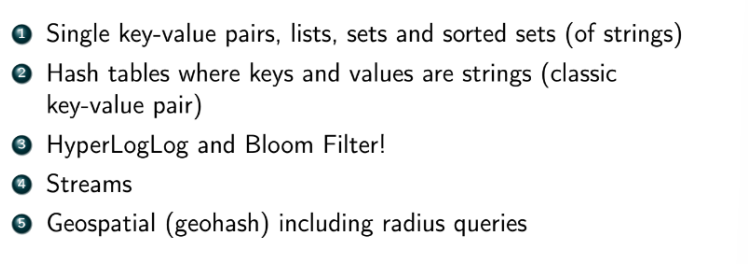
- we typically process data:
  - over a particular window of time
  - using an update rule if the full window does not fit in the buffer
- computations can be exact or approximate
- with respect to time
  - **tumbling window**: non-overlapping windows of time, consecutive
  - **hopping window** (moving average): fixed length period, but time periods overlap
- with respect to events
  - **sliding window**: constructed with some time period around each event
  - **session window**: defines an entire session from first event of particular type to last event (inefficient, similar to RDBMS)
- straggler events: produced in window w but does not arrived until after w has been closed an processed. What can we do?
  - ignore stragglers
  - later offer a correction
    - requires that aggregate is updatable, easily
- **bloom filter**
  - 

  - 

- **HyperLogLog**: similar to bloom filter, but we treat bitstring as a binary number and use run lengths and probability to provide an estimate of the number of distinct objects in a set

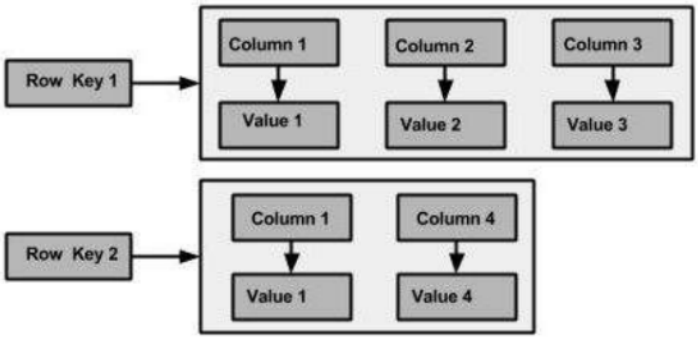
NoSQL

- Two different concepts:
  - a database where DML and DDL is something other than SQL (we prefer “not only SQL” instead of “no SQL”)
  - a database system that doesn’t use the relational model
  - common models:
    - key-value store
    - columnar
    - document store
    - graph
- most NoSQL systems are designed to be distributed. They make use of
  - replication (multiple copies of data distributed throughout system
  - sharding: different partitions of data live on different nodes
- **CAP theorem**
  - 

  - *it is impossible for both **C** and **A** to be satisfied *when a network partition or failure occurs in a distributed data system*
  - **P** must always be satisfied in a distributed system
- **Redis**: key-value store, but can be used as data structure store
  - data types:
    - 

  - persistence achieved through **snapshotting** from RAM to disk or a journal that records operations performed on data in RAM
  - Redis uses:
    - session caching
    - message queues
    - leaderboards
    - fast lookup and indexing
  - CAP is irrelevant to Redis since Redis isn’t distributed
- **Columnar databases**
  - fixed number of columns

    - each row contains different set of those columns (each row can have different numbers of columns)
    - **Cassandra**



        -
        - cannot use joins
        - designed to be unnormalized for fast lookups (used in data warehouses)
        - no foreign key, but primary keys do exist
        - rows are strictly ordered
        - uses **CQL (Cassandra Query Language)**
        - eventually consistent
        - Cassandra is AP
- **document store**
    - **MongoDB**: multiple databases, within each database there are multiple collections of documents. Documents are schemaless, there is no requirement that documents follow a strict schema
        - each document is in a JSON-like format called BSON
        - records -> collections -> databases
        - use *find* or *findOne* to find data

```
db.Books.find(
    // Selection condition (WHERE). Find programming books.
    { category: "programming" },
    // Projection operator. Hide _id, show title (0/1)
    { _id: 0,  title: 1 }
    // sort by title ascending (vs -1 descending), and limit.
).sort( { title: 1 } ).limit(5);
```

        -

Aggregations in MongoDB are a little weird:

```
db.Books.aggregate( [
  // Match stage (WHERE).
  {
      "$match": { category: "programming" }
  },
  // Group stage (GROUP BY). Can use multiple functions.
  // use _id: null for global aggregate.
  {
      "$group": { _id: "$language", AvgPrice: { "$avg": "$price" } }
  },
  //optional Projection
  { $project: { _id: 1, AvgPrice: 1} },
  // optional Sort stage
  { $sort : { price: -1 } },
  // optional Limit stage
  { $limit : 10 }
] );
```

■

MongoDB: Other Differences from RDBMS

Some other differences:

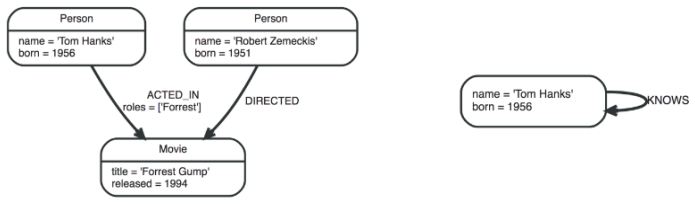
- **Data format:** Data is basically just a big JSON blob with no particular format or schema on disk.
- **File storage:** MongoDB can be used as a file system called GridFS. I suspect this is to allow functionality for something similar to Hadoop's HDFS tor a big data processing system released at a later date.
- **Aggregation:** is supported like in SQL, but can use internal **MapReduce** to distribute embarrassingly parallel computations.

■

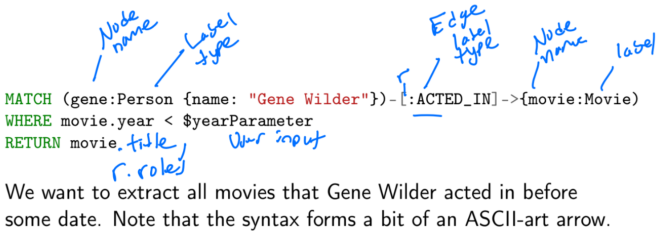
- ■ ensures strong consistency
- ■ uses horizontal sharding
- ■ by default, MongoDB is CP
- ■ Mongoose is a popular ODM for MongoDB

- **Graph store**
  - data stored as node or edge
  - nodes represent entities/relations
  - edges represent foreign keys/relationships
  - nodes and edges can have properties and labels
  - edges are usually directed but can be traversed in either direction

Example of Movie Data in neo4j

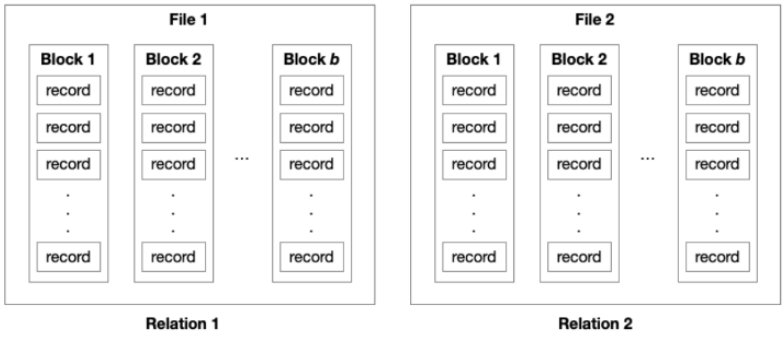


  - **neo4j**
    - ■ uses Cypher with MATCH and WHERE constructs

- MATCH: specifies a node pattern for pattern matching
- WHERE: similar to SQL WHERE, but applies to properties of nodes
- 

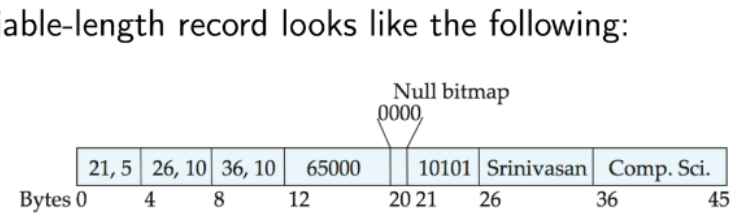
  - supports but does not require schemas
  - traversing the graph is the equivalent of executing a join
  - CAP theorem doesn't apply to neo4j

Data storage

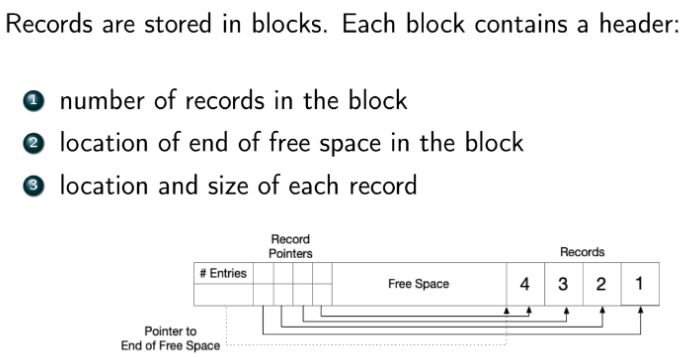
- **storage engine/storage manager**: responsible for storing, retrieving, and managing data in memory and on disk
  - also handles swapping data between disk RAM
- we choose how to store data on a system based on
  - speed of access
  - cost per unit of data
  - data reliability
- disk performance
  - access time: time from r/w request to time that data transfer begins
    - seek time + transfer time
  - seek time: time for head to move across platter to a sector
  - rotational latency: on in a track, how long for sector to be under head
  - data transfer rate: time to transfer data in sector from disk to main memory
- data on disk is addressed by a **block number**. Data is transferred between RAM and disk in term of **blocks** (pages)
- SSD's optimize for random access, HDD's are optimized for sequential access
- **file organization**
  - 

- fixed-length records
    - easy to process, wastes space
    - overflow block:

    -
- variable-length records
    - organizing record into a file:



    -
        - table of contents pointing to variable sized attributes, then data for fixed size attributes, then null bitmap, then data for variable sized attributes
    - space efficient, pain to process
    - organizing record into block:



    -
- **Index**

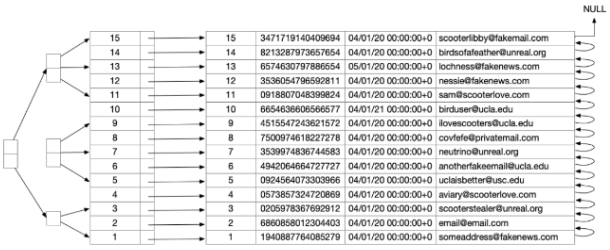
- given a record key, we can retrieve a **block number** and a **byte offset** within that block
- **ordered indices**: based on a sorted ordering of the sort-key values (sequential access)
  - **primary (clustering) index**: index entries sorted in the same method/order as the underlying data file
    - + block reads are minimized
    - + can perform range queries
    - - difficult to maintain contiguity on disk
  - **secondary (non-clustering) index**: data file is sorted by a different key/attribute than the index
    - must be dense
    - less efficient than primary indices
    - often used when there is no primary key
    - may need overflow block
- **hash indices**: based on a uniform distribution of values across a range of buckets (random access)
- **dense index**: every key that appears in a record also appears in the index
- **sparse index**: only a subset of the keys that appear in records appear int he index
  - usually only the first key in each block appears
- can have multiple indexes on a file
- primary index not required
- may have 2nd index on foreign, join, or unique key

Which type of index we use depends on several factors:

1. **Access Type:** equality, or range search?
2. **Access Time:** the time is takes to find a particular record
3. **Insertion Time:** time to insert records
4. **Deletion Time:** time to delete records
5. **Space overhead:** space the index data structure requires.

We look up records using a search key. This may be a a primary key, candidate key or superkey but not necessarily. The term key is different than we have been using.

-
- **B+ Tree**

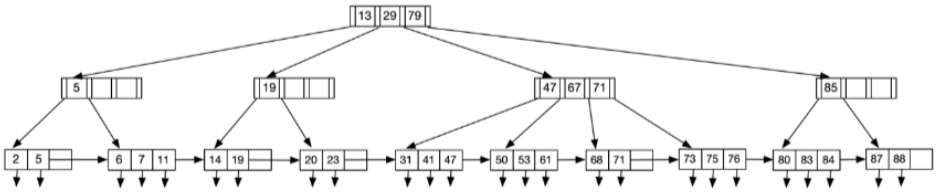


-
- uses divide and conquer

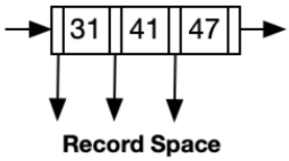
A B+ Tree builds a *multilevel* index as a multiway tree:

1. balanced, self-balancing *n*-ary tree
2. requires that all paths have the same length from root to leaf
3. contains a root, internal nodes, and leaves.
4. the root is either a leaf, or has 2 or more children.
5. has high fanout (n) to reduce the number of block reads
6. represents one block read at each level of the tree

-
- Uses:
  - Filesystems use B+ trees to index metadata about files
  - Key/value stores use B+ trees for data access

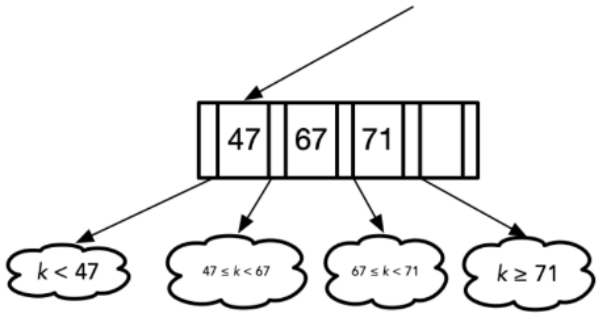


  -
    - bottom layer consists of keys that point to records on disk
- leaf node contains n pointers and n-1 key-values



For a B+ tree to retain its signature structure, a leaf node must contain at least $\lceil \frac{n-1}{2} \rceil$ key-values and at most $n-1$. Leaf nodes form a **dense** index.

  -



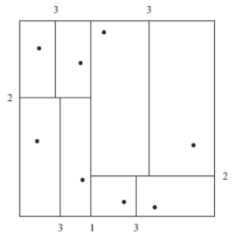
-
- non-leaf nodes for a sparse index on leaf nodes, holding up to n pointers and must contain at least ceiling(n/2) pointers
- slow insertion/deletion/query time, but fast rebuild time
- given height of tree = h, time to find key is $(h+1)(t_s+t_t)$
  - t_s = seek time
  - t_t = block transfer time
- range query: find keys between a and b
- *pointer collection method*: traverse all leaf nodes in range
- *record collection method*: go all the way down to record space before traversing all records in range
- **cons of B+ trees**
  - may have duplicate keys
    - solution: merge non-unique key with PK
  - variable length attributes, like strings, make it difficult to store in a tree
    - solution: use something else
  - bulk-loading arbitrary data into B+ tree is very inefficient
    - solution: load data first and then build tree

- **Indices in SQL**

```
CREATE INDEX  highway_lookup  ON caltrans
[ USING (btree | hash | gist | spgist | gin | brin) ]
(
    attribute1 [ASC | DESC] [NULLS {FIRST | LAST }],
    ...
);
```

- **hash indices**
  - good at equality queries, bad at range queries
  - cons:
    - insufficient buckets
    - bucket skew
  - static hashing: assume 1.2x the number of records you're going to have
  - dynamic hashing
- **indexing spatial data**
  - k-dimensional tree can be used to represent spatial data

In the 2D plane (or even 3D), we can use a special tree called a *k-d* tree, or *k*-dimensional tree. Each boundary line in this diagram represents one node in a *k-d* tree.



- temporal data can also be indexed

Query processing

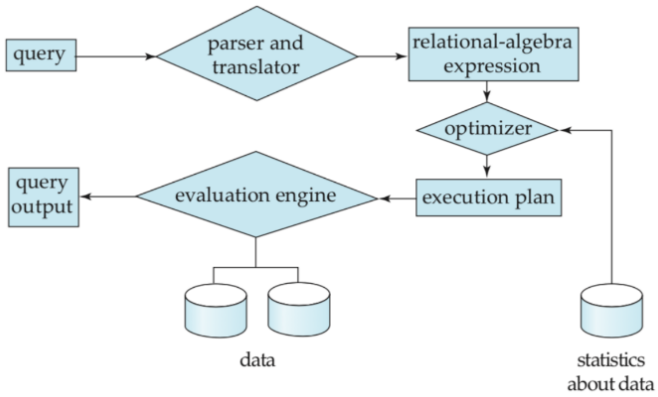
- results of query processing
  - parsing and translating -> expressions that can be used at physical level of RDBMS
    - similar to parser in a compiler
    - ensuring syntax is correct
    - translates to an internal form that is based on relational algebra
    - e.g.

Several different queries may lead to the same exact resultset and a query can be converted into relational algebra in many ways.

```
SELECT user_id
FROM bird_monthly_bill
WHERE amount > 1000;
```

1. $\sigma_{amount>1000}\left(\Pi_{user\_id,amount}\left(bird\_monthly\_bill\right)\right)$
2. $\Pi_{user\_id,amount}\left(\sigma_{amount>1000}\left(bird\_monthly\_bill\right)\right)$

  - optimization -> a series of query-optimizing transformations
    - picks most efficient query plan
      - criteria:
        - # of block transfers and seeks
        - # of tuples
        - current CPU/RAM usage
        - data transfer speed
        - disk space
        - network transfer
        - time
  - evaluation -> an actual evaluation of queries



-

  Suppose we have a sequential file, without an index, and we want to find records that match on some attribute that is **not** a key and there is no index.

  - b_r is the number of blocks in the file
  - t_T is the amount of time to transfer one block
  - t_S is the seek time.

  The total execution time is

  $$t_S + b_r \cdot t_T$$

-
- if blocks are not contiguous, we must seek for each block as well
  - $t_s b_r + t_T b_r$

- nested-loop join: given n_r x n_s pair of tuples, with b_r and b_s being the number of blocks in R and S:
  - O(n^2)
  - number of block transfers: $b_r + n_r b_s$
  - number of seeks: $b_r + n_r$
  - if one of the relations fits fully in memory, it should be in the inner loop
    - if R fits in memory:
      - if R is inner loop:
        - number of block transfers: $b_s + b_r$
        - seeks: 2
      - if R is outer loop:
        - number of block transfers: $b_r + n_r b_s$
        - seeks: $n_r + 1$
- block nested-join loop
  - number of block reads: $b_r(b_s + 1)$
  - number of seeks: $2b_r$
- indexed nested-join loop

Thus, scanning just R requires $b_r\,(t_T + t_S)$ time.

Now let's consider the index probes.

For each tuple in R, we lookup its key in some index. The total I/O time spent is cn_r, where c is the disk cost of a select from the index.

So the total time needed for the index nested loop join is

$$b_r\,(t_T + t_S) + cn_r$$

- merge join
  - R and S must be already sorted on some join key
  - number of block reads: $b_r + b_s$
  - number of seeks: proportional to ceiling($b_r/b_b + b_s/b_b$)
    - b_b: number of blocks that fit in buffer
- hash join

We use whichever relation (let's say it's R) is chosen to be the build side as follows:

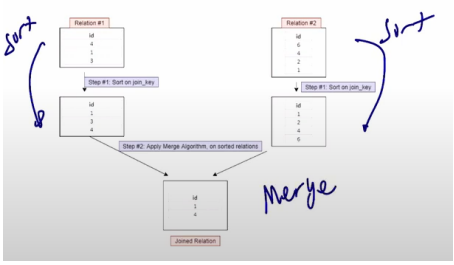
1. The system reads in as many blocks from R as will fit into the buffer and creates a hash table.
   - The key is the join key from R.
   - The value contains all of the other attributes.
2. We perform a full table scan on S (the **probe side**).
3. For each tuple in S, if the join key is contained in the build side hash table, output the match.
4. Once we are finished scanning S, we clear the hash table/build side, and start with step 1 again until all blocks of R are exhausted.

  - similar to block nested join loop
- indexed nested-loop, merge join, and hash join can only be used for simple join conditions (natural joins or equijoins)
- **Spark join algorithms**
  - shuffle based joins: if neither relation fits in RAM, we use this.

Shuffling works just like it does in the partition phase in Hadoop Map-Reduce. Each row of R and S is shuffled across the network to a partition such that all rows with the same value on the join key ends up in the same partition.

The join $R \bowtie_\theta S$ is then computed locally and results written to disk.

    - hash join
      - neither table needs to fit in RAM
      - doesn't require keys to be sortable
      - only supports equijoins, cannot perform full outer join
      - shuffling is expensive, hash table uses more RAM

- sort-merge join
  - keys must be sortable
  - susceptible to key skew
  - 

  - default join on Spark
- explain() will show query plan in Spark

- broadcast based joins: if one of the tables involved can fit entirely in RAM, the table is broadcasted to all executors in the cluster
  - broadcast nested-loop join: can handle all join types, typically least performant
  - broadcast hash join: only supports equijoins, cannot perform full outer join

In most practical cases, Spark will Sort-Merge Join (default) or Broadcast Hash Join if conditions allow. More specifically, Spark chooses a join algorithm using the following questions:

1. equi- or non-equi join?
2. did the user specify a hint?
3. was the hint actually applicable? If so, use it. If not, proceed in the following order:
   1. equi-join: broadcast hash, shuffle hash, sort merge (if sortable), Cartesian product, broadcast nested loop (and hope for the best).
   2. non-equi-join: broadcast nested loop join, Cartesian product (if inner-like), fallback to try broadcast nested loop join again.

The easiest thing to do is tell Spark what you want – you know the data size and the cluster architecture better than Spark.

- be explicit about broadcast join:

```
from pyspark.sql.functions import broadcast
...
large_df.join(broadcast(small_df), ["some_join_key"])
```

Or in Spark SQL:

```
SELECT /*+  BROADCASTJOIN(small) */ *
FROM large JOIN small
ON large.foo = small.foo
```

BROADCASTJOIN can be replaced with BROADCAST or MAPJOIN.

Transactions

- START TRANSACTION
  …
  COMMIT;

    - Everything between START TRANSACTION and COMMIT happens atomically (all or nothing is executed)
- autocommit: when enabled, there's an implied COMMIT after each statement

    To turn auto-commit on and off in various systems

System	Enable/Disable
PostgreSQL (via `psql`)	`\set AUTOCOMMIT on/off`
PostgreSQL (via `psycopg2`)	`connection.set_session(autocommit=True/False)`
MySQL	`SET autocommit=1/0`

    SQLite infers what the user wants. A series of operations within `begin transaction ... commit transaction` is treated as auto-commit off. Individual statements are treated as auto-commit on.

    -

    During a transaction, the RDBMS will start executing data operations. At any point a failure can occur:

    - data integrity failure
    - constraint failure
    - other errors (i.e. division by zero)
    - disk failure
    - system outage
    - etc.

-
    - when an error occurs, the transaction stops and performs a **rollback** on the transaction
        - similar to journaled filesystem
    - partially committed: written to buffer but not yet disk

Transactions are **ACID** compliant if they guarantee

1. **Atomicity.** The result of *all* operations in a transaction are represented in the database, or none are. Period.
2. **Consistency.** When transactions are executed in isolation, data remains consistent.
3. **Isolation.** When transactions are executed concurrently, none of the transactions need to know about each other, and none of them depend on each other. A transaction system must make this true.
4. **Durability.** After a transaction completes *successfully* the changes it makes persist to the database, **even if there are system failures.**

-
    - NoSQL cannot completely satisfy ACID
        - NoSQL satisfies ACID to varying degrees, some are fully ACID compliant as long as a transaction only touches one node

A matter of fact, NoSQL transactions are often stated to adhere to **BASE**:

Basically **A**vailable **S**oft-state **E**ventually consistent

ACID provides high consistency, BASE provides high availability.

-

1. **Basically Available.** Ensure availability of data by spreading and replicating it across the nodes of the database cluster.
2. **Soft State.** Due to the lack of immediate consistency, data values may change over time. Consistency is delegated to the application developer rather than the database.
3. **Eventually Consistent.** Immediate consistency is not guaranteed, but over time, data may become consistent.

-
    - tradeoffs
        - high isolation = less concurrency = worse performance = better consistency
- serial schedules are always consistent
    - if a concurrent schedule provides same result as serial schedule, then the concurrent schedule is consistent
- serializability

    - given a concurrent schedule, if we are able to swap instructions in both transactions until both transactions occur serially, then we have **conflict serializability**. *This means that the concurrent schedule is guaranteed to achieve consistency*
        - read and read on the same datapoint can be swapped
        - write and read on the same datapoint cannot be swapped
        - write and write on the same datapoint cannot be swapped

Consider some schedule S. We create a **precedence graph** G from S. The vertices consist of the individual transactions T_i, and we construct an edge from T_i to T_j if:

1. T_i executes `write(X)` before T_j executes `read(X)`.
2. T_i executes `read(X)` before T_j executes `write(X)`
3. T_i executes `write(X)` before T_j executes `write(X)`

Here we focus on *conflicting* operations that cross transactions rather than compatible operations like in the swap method. We have an edge $T_i \rightarrow T_j$ if T_i executes an operation that later conflicts in T_j.

    -
        - if a cycle forms in the graph, consistency is not guaranteed
        - otherwise, the schedule is conflict serializable
            - topo sort to get order of transactions in serial schedule that would give same result as current schedule
- **levels of isolation (strongest to weakest)**
    - serializable: schedule equivalent in result to a serial schedule
    - repeatable read: don't allow T_i to write to X if T_j re-reads that value later
    - read committed: T_i can only read data from T_j that has been committed
    - read uncommitted: T_i can read any of T_j's data even if it hasn't been committed
- **dirty read**: when a transaction can read uncommitted changes from other transactions
- **non-repeatable/fuzzy read**: when one transaction read a value, then some other transaction overwrites that value and commits it, then the first transaction read the value again
- **phantom read**: when one transaction inserts or deletes rows on a table in between two fetches in another transaction
    - applies to changes to a set of records, not the existing records themselves

	Dirty Read	Non-Repeatable Read	Phantom
Serializable	No	No	No
Repeatable Read	No	No	May Occur
Read Committed	No	May Occur	May Occur
Read Uncommitted	May Occur	May Occur	May Occur

-

- Locking
    - two modes

        - shared: allows read access to X for transaction, but no writes, transaction can have multiple shared locks on X and multiple transactions can have s-lock on X
            - multiple transactions can hold a shared lock on the same data item
        - exclusive: allows R/W access
            - only one transaction can hold an exclusive lock at a time (if a transaction holds an exclusive lock on A, no other transaction can hold either an exclusive or shared lock on A)
    - if we unlock too early, we may lose serializability
    - if we unlock too late, we may get deadlock
    - **two phase locking protocol (2PL):**
        - growing phase: transaction can acquire locks but can't release them
        - shrinking phase: transaction can unlock but cannot acquire more locks
        - ensures serializability
        - doesn't prevent deadlock
    - **deadlock**

        1. Transaction T_1 locks data point P with an exclusive lock.
        2. It then requests to lock data point Q, but T_2 has an exclusive lock on Q. So T_1 blocks.
        3. While T_2 holds the exclusive lock on Q, it requests a lock on P. But since T_1 already has an exclusive lock on P, T_2 blocks.

        - There are a few approaches to **preventing** deadlocks:

            1. We can acquire ALL required locks at once..
            2. Acquire locks in a **certain order** (say a topological sort) that prevents cycles in the wait-for graph.
            3. **Use preemption.** If T_i requests a lock that T_j holds, we preempt, rollback and then grant the lock to T_i based on several criteria.
            4. **Lock timeouts.** If a lock waits and is not acquired within n seconds, the transaction rolls back and starts over. Very limited.

- Redis only truly supports isolation in ACID
- consistency and isolation of MongoDB has been disputed

For exams:

We want to test if you really understand how to create the proper join conditions, so I am picky on how you write joins for the exam:

1. Never use `NATURAL JOIN` in SQL, only use it for relational algebra if needed.
2. Do not use `USING` to replace the `ON` clause (discussed in the book).
3. Avoid using the `SELECT FROM t1, t2, ... WHERE` method. It's error prone and unreadable.
4. Avoid using anything else that does not require your to clearly and explicitly specify the join condition.
5. Only use column names in `GROUP BY` and `ORDER BY` **not** column sequence numbers.
6. A calculated aggregation should never be the first column in the output.

- Subqueries are sometimes unavoidable, but minimize the number of subqueries you use in your responses
  - if the problem involves HAVING, use it
  - do not use CTEs to avoid a subquery (no WITH)
- potential questions about functional dependencies
  - find FDs given table
  - use FDs to determine normal form
- enums:

  Some important points about `ENUM`s:

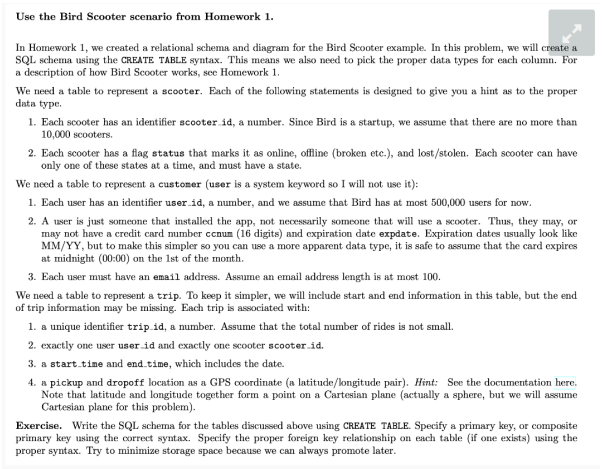
  1. Order of the attributes is defined when the type is constructed.
  2. If not careful, we can have nonsense comparators 'mad' > 'sad' and we can use functions like `MIN` and `MAX`.
  3. Values are case sensitive and whitespace matters.
  4. Can add new values and rename values. Cannot delete or reorder.
  5. Good for saving space since we don't need to use `character` or `varchar`.
  6. Don't need to maintain a mapping of integers to values.
  7. 4 bytes

  -

  1. Use `HAVING` when filtering using an aggregation, not `WHERE`.
  2. When writing a query, `WHERE` comes before `GROUP BY` comes before `HAVING` comes before `ORDER BY`.
  3. Column aliases can be used in `ORDER BY` but not in `WHERE` or `HAVING`.

-

Extra homework stuff



Use the Bird Scooter scenario from Homework 1.

In Homework 1, we created a relational schema and diagram for the Bird Scooter example. In this problem, we will create a SQL schema using the `CREATE TABLE` syntax. This means we also need to pick the proper data types for each column. For a description of how Bird Scooter works, see Homework 1.

We need a table to represent a `scooter`. Each of the following statements is designed to give you a hint as to the proper data type.

1. Each scooter has an identifier `scooter_id`, a number. Since Bird is a startup, we assume that there are no more than 10,000 scooters.
2. Each scooter has a flag `status` that marks it as online, offline (broken etc.), and lost/stolen. Each scooter can have only one of these states at a time, and must have a state.

We need a table to represent a `customer` (`user` is a system keyword so I will not use it):

1. Each user has an identifier `user_id`, a number, and we assume that Bird has at most 500,000 users for now.
2. A user is just someone that installed the app, not necessarily someone that will use a scooter. Thus, they may, or may not have a credit card number `ccnum` (16 digits) and expiration date `expdate`. Expiration dates usually look like MM/YY, but to make this simpler so you can use a more apparent data type, it is safe to assume that the card expires at midnight (00:00) on the 1st of the month.
3. Each user must have an `email` address. Assume an email address length is at most 100.

We need a table to represent a `trip`. To keep it simpler, we will include start and end information in this table, but the end of trip information may be missing. Each trip is associated with:

1. a unique identifier `trip_id`, a number. Assume that the total number of rides is not small.
2. exactly one user `user_id` and exactly one scooter `scooter_id`.
3. a `start_time` and `end_time`, which includes the date.
4. a `pickup` and `dropoff` location as a GPS coordinate (a latitude/longitude pair). *Hint:* See the documentation here. Note that latitude and longitude together form a point on a Cartesian plane (actually a sphere, but we will assume Cartesian plane for this problem).

Exercise. Write the SQL schema for the tables discussed above using `CREATE TABLE`. Specify a primary key, or composite primary key using the correct syntax. Specify the proper foreign key relationship on each table (if one exists) using the proper syntax. Try to minimize storage space because we can always promote later.

```
CREATE TYPE status_setting AS ENUM ('online', 'offline', 'lost/stolen');

CREATE TABLE  scooter (
    scooter_id smallint,
    status status_setting NOT NULL,
    PRIMARY KEY (scooter_id)
);

CREATE TABLE customer (
    user_id integer,
    ccnum numeric(16),
    expdate date,
    email varchar(100) NOT NULL,
    PRIMARY KEY (user_id)
);

CREATE TABLE trip (
    trip_id bigint,
    user_id integer NOT NULL,
    scooter_id smallint NOT NULL,
    start_time timestamptz NOT NULL,
    end_time timestamptz,
    pickup point NOT NULL,
    dropoff point,
    PRIMARY KEY (trip_id),
    FOREIGN KEY (user_id) REFERENCES customer(user_id),
    FOREIGN KEY (scooter_id) REFERENCES scooter(scooter_id)
);
```

Write a query that computes the total number of trips that started at each BART station. Only keep stations that had more than 1,000,000 boardings. Output the station Name, Location and total Throughput as total throughput.

Submit only the query. If possible, please use Courier/Teletype font.

```
SELECT
    Name,
    Location,
    SUM(Throughput) AS total_throughput
FROM ridecount
JOIN station ON ridecount.Origin = station.Abbreviation
GROUP BY Name, Location
HAVING SUM(Throughput) > 1000000;
```

Write a query that computes movement among cities. That is, compute the total number of trips (Throughput) between city A and city B and call it total rides. For example, we want to know how many trips started in Oakland and ended in Fremont, and vice verse. Output the city (Location) corresponding to Origin, the city Location corresponding to Destination and total rides. Do not output Origin or Destination, as that would not make sense. Sort from largest to smallest total rides.

Submit only the query. If possible, please use Courier/Teletype font.

```
SELECT
    o_station.Location AS origin_location,
    d_station.Location AS dest_location,
    SUM(Throughput) AS total_rides
FROM ridecount JOIN station o_station
ON ridecount.Origin = o_station.Abbreviation
JOIN station d_station ON ridecount.Destination = d_station.Abbreviation
GROUP BY o_station.Location, d_station.Location
ORDER BY total_rides DESC;
```

Write a query that finds the maximum Throughput ever recorded in this dataset, as well as the Origin and Destination for this trip, and the date/time (Tstamp). How can we do it without using LIMIT and ORDER BY?

Submit only the query. If possible, please use Courier/teletype font. *If you wish to explicitly answer the last part of the question, use a SQL comment. Start a new line starting with --*

```
SELECT
    Origin,
    Destination,
    Tstamp,
    Throughput
FROM ridecount
WHERE Throughput =
(
    SELECT
        MAX(Throughput)
    FROM ridecount
);
-- To avoid using LIMIT or ORDER BY, we can use a subquery as shown above
```

Write a query that computes two new columns: (1) the elapsed time (in minutes) of each trip, and (2) the total cost of the trip. The trip charge is computed as follows: $1 flat rate for each trip plus $0.15 per minute. Fractional minutes should be rounded up (ceiling), so 4.02 minutes becomes 5 minutes. If the trip does not have an end time (scooter was stolen etc.), the length of the trip shall be 24 hours (1440 minutes) and the user should be charged based on 24 hours of use. Your results should include the trip id, user id, trip length and trip cost. There are at least two ways to do this problem. We prefer the method with the subquery and/or join. Compute the length of each trip first, and then compute the cost. Order the results by trip id in ascending order. Also, submit the top 10 rows of your output, without any special ordering.

Submit your SQL query in the first box. If possible, please use Courier/teletype font.

```
SELECT
    user_id,
    trip_id,
    trip_length,
    1 + 0.15 * trip_length AS trip_charge
FROM
    (
        SELECT
            trip_start.user_id AS user_id,
            trip_start.trip_id AS trip_id,
            COALESCE(
                CEILING((
                EXTRACT(EPOCH
                FROM trip_end.time) -
                EXTRACT(EPOCH
                FROM trip_start.time))
                / 60),
                1440) AS trip_length
        FROM trip_start
        LEFT JOIN trip_end
        ON trip_start.trip_id = trip_end.trip_id
    ) _
    ORDER BY trip_id;
```

Modify your query so that it computes the total amount that each user has spent on Bird Scooter. Again, use your previous response and assume that we did not store the intermediate result from the previous part. Report the user id and the total amount spent as total spent. Sort by user id in ascending order. Be careful here.

Submit only your query. If possible, please use Courier/teletype font. Note that you may not reference any previous written queries with placeholders. The query must be complete.

```
SELECT
    user_id,
    SUM(1 + 0.15 * trip_length) AS total_spent
FROM
    (
        SELECT
            trip_start.user_id AS user_id,
            trip_start.trip_id AS trip_id,
            COALESCE(
                CEILING((
                EXTRACT(EPOCH
                FROM trip_end.time) -
                EXTRACT(EPOCH
                FROM trip_start.time))
                / 60),
                1440) AS trip_length
        FROM trip_start
        LEFT JOIN trip_end
        ON trip_start.trip_id = trip_end.trip_id
    ) _
GROUP BY user_id
ORDER BY user_id;
```

- EXISTS, NOT EXISTS, EXCEPT, IN, NOT IN

- Exists:

  The `EXISTS` operator is used to test for the existence of any record in a subquery.

  The `EXISTS` operator returns TRUE if the subquery returns one or more records.

  EXISTS Syntax

  ```
  SELECT column_name(s)
  FROM table_name
  WHERE EXISTS
  (SELECT column_name FROM table_name WHERE condition);
  ```

- Except:
  - The SQL EXCEPT statement returns those records from the left SELECT query, that are not present in the results returned by the SELECT query on the right side of the EXCEPT statement.

    ```
    SELECT column1 [, column2 ]
    FROM table1 [, table2 ]
    [WHERE condition]

    EXCEPT

    SELECT column1 [, column2 ]
    FROM table1 [, table2 ]
    [WHERE condition]
    ```

- In:

  The `IN` operator allows you to specify multiple values in a `WHERE` clause.

  The `IN` operator is a shorthand for multiple `OR` conditions.

  IN Syntax

  ```
  SELECT column_name(s)
  FROM table_name
  WHERE column_name IN (value1, value2, ...);
  ```

Redis Demo

```
# Set a key
SET foo 1

# Increment the value
INCR foo                (good for counters)
GET foo

# Retrieve a missing key.
GET bar

# Increment by another value
INCRBY foo 100
GET foo

# Expire this key/value pair after 1 hour.
EXPIRE foo 60                   # Great for caching.

# Does the key exist?
EXISTS foo     (returns 1 or 0)

# Get the value for key
GET foo


# Set values for multiple keys
MSET bar 42 baz 11

# Get values for multiple keys.
MGET bar baz          # (nil) is none

# Let's check up on our cached key.
GET FOO

##### Strings

# Create a key/value pair, where value is a string.
SET username ryanrosario
GETRANGE username 4 11          # indexed from 0. Same as SUBSTR
APPEND username _teaches_cs143
GET username
STRLEN username
```

```
# List values.
# Doesn't have to be defined beforehand
LPUSH words man
LPUSH words woman
LPUSH words person
RPUSH words camera
RPUSH words tv

$ Get the list length.
LLEN words

# Get the words in order.
LRANGE words 0 4

# Pop from either side of the list. Can use as a dequeue, queue or stack.
LPOP words
RPOP words
RPOP words
RPOP words
RPOP words
RPOP words   # no error

# Sets
# Tropical fruits in smoothie 1
SADD fruits1 coconut
SADD fruits1 pineapple guava avocado

# Tropical fruits in smoothie 2
SADD fruits2 pitaya orange guava pineapple

# What fruits are in both?
SINTER fruits1 fruits2

# What fruits do I need to buy to make both?
SUNION fruits1 fruits2

# Shopping list
SUNIONSTORE shopping fruits1 fruits2

# Get all members
SMEMBERS shopping

# Get a random set element
```

SRANDMEMBER shopping

Create a hash (dictionary)
HSET professor firstname ryan lastname rosario university ucla
HGET professor university

List all the keys
KEYS *

Delete some keys we created
DEL foo username counts fruits

Clean up
FLUSHDB
FLUSHALL

MongoDB Demo

```
/*CS143 - WINTER 2023 - LECTURE 13 - MONGODB DEMO

To install MongoDB, follow the directions below:
https://docs.mongodb.com/manual/installation/

BASIC CRUD (WITHOUT THE R)
*/

show dbs

use demo          // doesn't need to already exist
show collections

//Insert one record
db.cs22SCourses.insertOne(
      {
            code: "CS 143",
            title: "Data Management Systems",  // String
            lec_days: ["M", "W"],  // Array
            time_from: 16,
            time_to: 18,
            professor: {
                  name: "Rosario, R. R.",
                  email: "rrosario@cs.ucla.edu"
            },
            ta: ["Tyler", "Song", "Vivian"],
      }
```

```
);

// Add two graders.
// Add a TA to the TA array.
db.cs22SCourses.updateOne(
      { code: "CS 143" },
      {
            $set: {
                  grader: ["Madhav", "Nicole"]
            },
            $push: { ta: "Hughes" }
      }
);

db.cs22SCourses.insertMany([
      {
            code: "CS 130",
            professor: ["Burns, M.", "Hyman, J."]
      },
      {
            code: "CS 111"
      }
]);

// Delete CS 111.
db.cs22SCourses.deleteOne({
      code: "CS 111"
});

db.cs22SCourses.deleteMany({});



/* YELP QUERYING DEMO

Though we will try to create a Docker container for you, especially if
there is a
homework using it.

Then open mongodb shell by typing "mongo".
Create the Yelp database
*/

use yelp

show collections
```

```
db.business.explain()

// What does a Yelp business record look like?
> db.business.findOne({ categories: {$regex: "Restaurants" } })
{
      "_id" : ObjectId("60b540bdd8f924bd13670997"),
      "business_id" : "NDuUMJfrWk52RA-H-OtrpA",
      "name" : "Bolt Fresh Bar",
      "address" : "1170 Queen Street W",
      "city" : "Toronto",
      "state" : "ON",
      "postal_code" : "M6J 1J5",
      "latitude" : 43.6428886,
      "longitude" : -79.4254291,
      "stars" : 3,
      "review_count" : 57,
      "is_open" : 1,
      "attributes" : {
            "WiFi" : "u'no'",
            "BikeParking" : "True",
            "RestaurantsPriceRange2" : "2",
            "BusinessParking" : "{'garage': False, 'street': True,
'validated': False, 'lot': False, 'valet': False}",
            "RestaurantsTakeOut" : "True",
            "Caters" : "False"
      },
      "categories" : "Juice Bars & Smoothies, Food, Restaurants, Fast
Food, Vegan",
      "hours" : {
            "Monday" : "8:0-21:0",
            "Tuesday" : "8:0-21:0",
            "Wednesday" : "8:0-21:0",
            "Thursday" : "8:0-21:0",
            "Friday" : "8:0-21:0",
            "Saturday" : "9:0-21:0",
            "Sunday" : "9:0-21:0"
      }
}

/* Can search a range using:
field : { $gte: value, $lte: value} */


// Let's retreive all Starbucks, their states and ratings.
db.business.find({
      name: "Starbucks",
}, {
```

```
      name: 1,
      state: 1,
      stars: 1,
      _id: 0
});

// To make it a "non equal", we can replace name: "Starbucks" with name: {
$ne: "Starbucks" }

// Let's retrieve all Indian, Mexican or Italian restaurants that are
either
// casual attire or takeout (if it's not casual attire, I will take the
food out).
// Additionally, if I sit inside, there should be free Wifi.
db.business.find({
      $or: [
            { "attributes.RestaurantsAttire":  "'casual'",
              "attributes.WiFi": "'free'" },
            { "attributes.RestaurantsTakeOut": "True" }
      ],
      categories: {$contains: "Restaurants"},
      categories: {$regex: "Indian|Italian|Mexican"},
}, {
      name: 1,
      categories: 1,
      stars: 1,
      state: 1,
      _id: 0
}).sort( { stars: -1 } );


// Average rating for Starbucks in the USA and Canada (which has the state
field).
db.business.aggregate(
      [
            {
                  $match: {
                        name: "Starbucks"
                  }
            },
            {
                  $group: {
                        _id: "$state",
                        average: { $avg: "$stars" }
                  }
      }
   ]
```

```
);

/* First Stage: The $match stage filters the documents by the name field
 and passes to the next stage those documents that have name equal
 to "Starbucks".

 Second Stage: The $group stage groups the documents by state field to
calculate
 the average star rating.
*/
```

Neo4j Cypher Demo

```
// Show entire graph
MATCH (n) RETURN n


// Find all moves made in the 90s.
MATCH (nineties:Movie) WHERE nineties.released >= 1990 AND nineties.released < 2000
RETURN nineties.title

// Find Tom Hanks movies that he acted in. Returns a Person node called tom,
// which is Tom Hanks info. The unknown result is called tomHanksMovies which
// must be of type Movie because :ACTED_IN always point Person->Movie
// Get the movie title, year it was released and his role in the movie.
MATCH (tom:Person {name: "Tom Hanks"})-[r:ACTED_IN]->(tomHanksMovies) RETURN
tomHanksMovies.title, r.roles, tomHanksMovies.released

// What about roles he played?

// Who directed Cloud Atlas?
// cloudAtlas name isn't used. directors is the unknown result.
MATCH (cloudAtlas {title: "Cloud Atlas"})<-[:DIRECTED]-(directors) RETURN directors.name
// cloudAtlas:Movie is optional since there is only one node with title Cloud Atlas

// Tom Hanks' co-actors
// We want to find all Person nodes that point to some Movie m that Tom Hanks also
// points to as an actor. The unknown output is coActors.
MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors)
RETURN coActors.name
// We don't need m:Movie because all PERSON-[:ACTED_IN]-> relationships yield a movie.

// What about all movies that Tom Hanks and Meg Ryan acted in?
```

```
// Find all people that worked on Cloud Atlas
// Returns a property of a Person node (name), the type of the relationship, and any
// any attributes
MATCH (people:Person)-[relatedTo]-(:Movie {title: "Cloud Atlas"}) RETURN people.name,
Type(relatedTo), relatedTo

// Note here that we don't need m:Movie for example. We can just use :Movie because
// we never use any of the attributes of the Movie node.

// Find all movies AND people between 1 and 4 hops away from Kevin Bacon
// hollywood is unknown output.
MATCH (bacon:Person {name:"Kevin Bacon"})-[*1..4]-(hollywood)
RETURN DISTINCT hollywood

// What about only people that ACTED_IN movies?

// Shortest path from Kevin Bacon to Meg Ryan
MATCH p=shortestPath(
(bacon:Person {name:"Kevin Bacon"})-[*]-(meg:Person {name:"Meg Ryan"})
)
RETURN p
```