Lecture 1 - Relational Models - 4/5/19

Dr. Ryan Rosario

Problems with Flat File System

- Data Integrity and Redundancy
  - Differing preferences formats
  - Languages among sysadmins lead to possible data inconsistencies
  - Duplicate Data
- Sysadmin is the Bottleneck: all system changes, course changes restrictions must be encoded in a custom application.
- Lack of Atomicity: system failures in the middle of a multi-step process can result in lost data. (i.e. bank transfer, exchanging classes). we should do it perfectly.
- Concurrent Access: It's possible for data to get out of sync when multiple people read/write concurrently
- Security: Without proper filesystem controls, flat files can be read by anybody with access to the filesystem.

Better Approach: Databases

- A database abstracts away how the data is stored, maintained and processed.
- Users don't necessarily care how the data is laid out on disk.
- Big Data and distributed systems, such as Spark, have reintroduced the importance of understanding how data is laid out on disks/nodes
- Database Purposes
  - Provides a way to view, add, update and delete data without worrying about files and breaking data integrity.
  - Provides ONE single location for all data in the database

Examples of Databases that You Wouldn't Expect

- Blockchain
  - Blockchain is similar to a distributed database, with one major difference.
  - With blockchain, 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.
  - Coindesk: >"Blockchains allow different parties that do not trust each other to share information without requiring a central administrator. Transactions are processed by a network of users acting as a consensus mechanism so that everyone is creating the same shared system of record simultaneously."
- Git
  - Git can be used to track any kind of content, and it can be used to create a NoSQL database.
  - Git is basically a key-value store.

Levels of Abstraction 1. Physical. How the data are stored. * MySQL has 9 storage engines, most importantly, MyISAMa and InnoDBb (speed, reads) and InnoDB (foreign keys and transactions). * PostgreSQL only has one. 2. Logical. * Describes what data there are and the relationships among the data. 3. View. * What the user sees. It is typically just a part of the database, such as used to generate a report. In the flat file case, the user relied on the sysadmin to "see" the data.

A database (more precisely a table or schema, which we will discuss in a bit) is a way of abstracting a structured type, like you'd see in C++.

```
struct Section
{
    int srs; // Unique ID number for this section
    int cap; // Max number of students allowed
    int instructor_uid; // Instructor's ID
    int parent_srs; // link to lecture
};
```

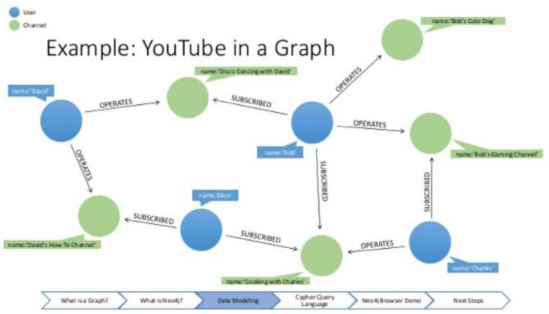
Instances and Schema

- The information stored in a database at a particular point in time is called an instance. This terminology has been resurrected by its use with Amazon Web Services.
- The overall design of a database is called a schema.
  - The term schema is used interchangeably to describe the logical structure of a database or a relation (table).
- A subschema refers to the design of a particular part of the database, (often still called schema) if it refers to a DB or a table.

Data Model

The data model contains "conceptual" tools for describing data, their relationships, semantics and consistency constraints. Contains 5 major types:

1. **Relational**: Uses a collection of tables to represent data and their relationships. Each table has columns with unique names and a data type. Each column represents an attribute, and each row represents a record.
2. **Entity-Relationship (ER)**. Uses a collection of basic objects called entities and relationships among them. Typically used to visualize a database design. 3.**Object-Oriented**: Draws a strong analogy to object-oriented programming with encapsulation, methods and object identity.
   - Data are essentially treated as instances of classes rather than tables.
3. **Document**: Individual data objects may have different sets of attributes.
   - Each data record mimics a flexible struct with no, or few, fixed fields.
   - Very different from the relational model, but many common semi-structured data types can be converted into the relational model.
   - JSON and XML are two examples of data types that are natural for the semi-structured model (Document Database).
4. **Network/Hierarchical/Graph**:
   - Actually pre-dates the relational model.
   - Defines data records as nodes, and relationships between records as edges.
   - Was considered unwieldy as they tied data very closely to database implementation details. This is still the case.
   - One notable NoSQL example is neo4j.



Database Languages

1. Data Definition Language (DDL)
   - A query is a written expression to retrieve or manipulate data.
   - A query language is simply the language it is written in. (Ex: SQL)
     - A SQL query takes takes one or a pair of relations and outputs a single relation as a result.
   - Note that JavaScript can be a query language!
   - **SQL**

    - Is not a procedural language.
    - There are computations that cannot be done in SQL, for example, sequential or iterative computations typically used in mathematical fields. Or algorithms that require the user to specify how to perform a computation.
    - In the case that SQL isn't enough, extract the data into your favorite language and then use that language to transform the data however you like, then insert back
2. Data Manipulation Language (DML)
    - Specifies a schema, a collection of attribute names and data types. It may also specify storage structure and access methods.
    - Constraints that can be specified
        1. **Domain Constraints**: restrict the values of a particular column to a particular type, or a particular set of values.
        2. **Referential Integrity**: There are many situations where a value in one relation must appear in some other relation for the data to be considered complete and valid. (Ex: Foriegn Keys)
        3. **Assertions**: Both a and b are types of assertions, but there are several other constraints we can impose. In the student records case, we can impose some example constraints that are checked by the DBMS on each attempted data manipulation:
        4. **Authorization**: Can restrict access to databases and schemas as well as particular operations on these databases and schemas. We can apply these rules to users, groups, etc.

Data Storage and Querying

- Databases have a storage manager that abstracts how the data is laid out on disk.
- Most of the data cannot fit in RAM, and must be read in from disk. This must be done eficiently because reading from the disk is very slow.
- The Storage Manager has the following duties
    1. **Authorization and data integrity checks** to prevent unauthorized access and enforce integrity constraints.
    2. **Transaction management** ensures the database remains in a consistent state despite a system failure, and that concurrent accesses are handled appropriately. A transaction is a series of operations that must be executed as a logical unit, in a particular order.
    3. **File manager** allocates and manages disk space.
    4. **Buffer manager** deals with swapping data from disk to RAM and back.
- The storage manager uses a few of its own data structures
    1. Data files which store the... well... data.
    2. Data dictionary containing metadata about the structure and schema of the database.
    3. Indices that enable optimized and eficient lookup of data.
- Another element is a query manager.
    - When a query is executed the DML (i.e. SQL) statements are organized into a query plan that consists of low-level instructions that the query evaluation engine understands

Note on Duplicate Data: Duplicate data should of course try to be avoided, this can be done through the process of deduplicating, called normalization

Distributed Architecture

- In a distributed architecture, there are several database servers. They may all contain identical data (replicated) or different parts of the data (sharded) that depend on geography or some other circumstance.

- Replicated or Sharded?

    1. You are developing a new social network to replace Facebook. It will allow people to create profiles and share content on a newsfeed across the entire (connected) world.
    2. You are the system administrator for UCLA and are responsible not only for student records, but also healthcare records at a variety of hospitals and medical centers in Southern California.
    3. You are creating a new blogging system that protects user privacy by relying on the user's own

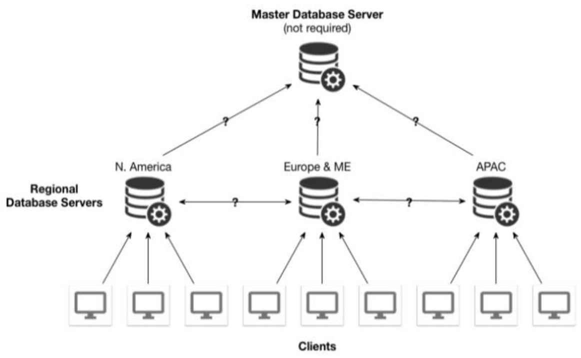


Figure 1: Distributed Architecture

system to host their content.

Relational Model

- A relational database consists of a collection of relations (often called tables).
- Each relation consists of records and attributes:
- Each row is a record, with a set of attributes that are related to that record.
- Each column represents a particular attribute, has a unique name, and a particular data type. (Columns are also referred to as n-tuples)
- Each attribute has a domain, a set of legal values. This domain can be discrete or continuous.
- An attribute can have a null value meaning it is unknown or does not exist.

SuperKey

A **superkey** is a set of one or more attributes that uniquely identifies a tuple and distinguishes it from all other tuples.

Illustration: Suppose we use the sequential comment id version of the youtube comment relation where the superkey K is video id and comment id.

1. I can also include other attributes and it is a valid superkey, but it contains redundant information because video id and comment id are the only attributes we need to uniquely identify a tuple.
2. Suppose we drop video id from K. comment id is a proper subset of K, but it is not a superkey because each video with at least one comment will have a duplicated comment id.

Thus, you can prove that comment id and video id are a composite superkey.

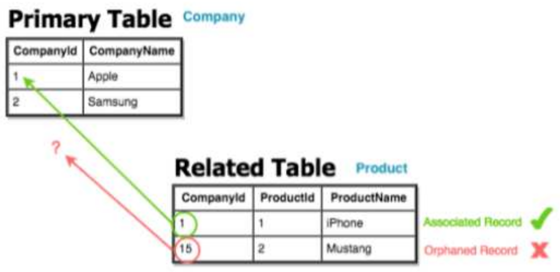
Primary Key

- Since a superkey uniquely identifies a tuple in a relation, no two tuples may same the same values for all of the attributes in K. A superkey that is minimal is also called a candidate key.
- A candidate key chosen by the database designer is more commonly called a primary key.
- A primary key MUST be unique, and MUST be `NOT NULL`. For composite primary keys, all components of the primary key must be `NOT NULL`.

Foriegn Key

- For a particular relation R, a foreign key is an attribute of R that is the primary key of some other relation R'
- A foreign key is an attribute that can be used to tie together the tuples of two relations together into one. A foreign key in R does not uniquely identify a tuple in R, but does in the referred relation R'.
- Foreign keys are often used to satisfy referential integrity constraints.
  - Suppose we have two relations: company that contains an ID and the name of a company, and product which maps products to the company that created it.

- It does not make sense for a product to not be associated with a company in this schema. This design violates referential integrity



CompanyId	CompanyName
1	Apple
2	Samsung

CompanyId	ProductId	ProductName
1	1	iPhone
15	2	Mustang

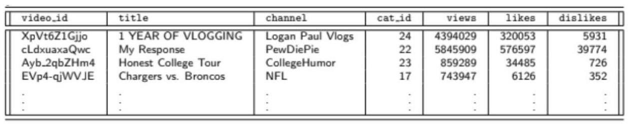
Relational Algebra

All relational databases provide a set of operations that can be performed on a single relation, or on a pair of relations. We will introduce most of them today, and see the rest next time. We can work with multiple relations, by chaining operations on pairs of relations. This is how we will do it in SQL, unsurprisingly.

Lecture 2 - Relational Algebra - 4/7/19

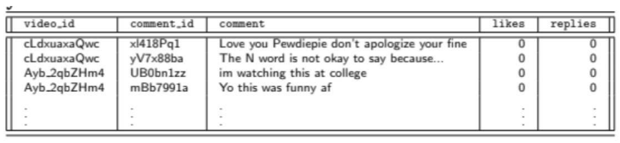
Relational algebra is a semantic system used for modeling the operations on relational data developed by Edgar F. Codd at IBM

For the following examples, we will use the following data:



video_id	title	channel	cat_id	views	likes	dislikes
XpVt6Z1Gjjo	1 YEAR OF VLOGGING	Logan Paul Vlogs	24	4394029	320053	5931
cLdxuaxaQwc	My Response	PewDiePie	22	5845909	576597	39774
Ayb_2qbZHm4	Honest College Tour	CollegeHumor	23	859289	34485	726
EVp4-qjWVJE	Chargers vs. Broncos	NFL	17	743947	6126	352
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Figure 1: youtube.video



video_id	comment_id	comment	likes	replies
cLdxuaxaQwc	xl418Pq1	Love you Pewdiepie don't apologize your fine	0	0
cLdxuaxaQwc	yV7x88ba	The N word is not okay to say because...	0	0
Ayb_2qbZHm4	UB0bn1zz	im watching this at college	0	0
Ayb_2qbZHm4	mBb7991a	Yo this was funny af	0	0
⋮	⋮	⋮	⋮	⋮

Figure 2: youtube.comment

Select σ

1. Select retrieves a subsete of tuples from a single relation that satisfies a particular constraint and returns a new relation that is a subset of the original version.
   - In relational algebra, σ_{ψ} retrieves a set of tuples that meet condition ψ.
   - Alternatively:
   $$\sigma_{\psi}(R) = \{t \in R : \psi(t)\} \tag{1}$$
   - Predicates appear in a subscript of σ and contain boolean expressions
   - If we want all videos that have been viewed over 1 million times, we can write this as:
   $$\sigma_{\text{views}>100000}(youtube\_video) \tag{2}$$
   - We can also use more complex predicates using conjunction/disjunction
   - We user $\neg$ to represent to specify the logical not operator.
   - An Example of select as a SQL implementation:
   ```
   SELECT title
   FROM youtube_videos
   WHERE likes > dislikes
   AND views > 1000000
   AND cat_id = 24
   ```
   - This can be written in relational algebra as:
   $$\sigma_{\text{likes}>\text{dislikes}\wedge\text{views}>1000000\wedge\text{cat\_id}=24}(youtube\_videos) \tag{3}$$
   - **Note**: the select operator extracts tuples not in attributes
   - **Note**: σ refers to `SQL WHERE` clause, not `SELECT`
2. Projection Π
   - It extracts attributes from a set of tuples
   - Given a relation R and an arbitrary tuple t and a subset of attributes $a_1 ... a_n$
   $$\Pi_{\text{a}_1,...,\text{a}_\text{n}}(R) = \{t[a_1, ..., a_n] : t \in R\} \tag{4}$$
   - The projection is typically the last (outermost) operation done on a relation

- Projection can be generalized to:
  1. Use functions like `MAX, MIN, COUNT` (Ex: $\Pi_{\text{AVG(likes)}}(R)$)
  2. Create new attributes or rename attributes using $\rightarrow$a notation
     - Ex: of combining columns to make a new one $\Pi_{\text{likes+dislikes}\rightarrow\text{interactions}}$
- The projection operator removes duplicates, making it similar to `SELECT DISTINCT`

3. Combining σ and Π
   - Can construct a projection from the `SELECT` clause then:

$$\Pi_{\text{title,channel}}(\sigma_{(\text{likes}>\text{dislikes})\wedge(\text{views}>1000000)}(youtube\_video)) \tag{5}$$

   - **Note**: conjunctions can be written as a chaining of multiple sets:

$$\Pi_g(\sigma_{\text{f}(\text{a}_1)}(...\sigma_{\text{f}(\text{a}_\text{n})}(R))) = \Pi_g(\sigma_{\text{f}(\text{a}_1)\wedge...\wedge\text{f}(\text{a}_\text{n})}(R)) \tag{6}$$

4. Cartesian Product x
   - The cartesian product combines tuples from two relations, in all possible combinations
   - This can formally be written as:

$$A \times B = \{(a, b) : a \in A, b \in B\} \tag{7}$$

   - If two relations, R_1, R_2 have attribute nums n_1, n_2 then

$$\|R_1 \times R_2\| = n_1 n_2 \tag{8}$$

   - Cartesian products are thus an $O(n^2)$ operation where $n = max(n_1, n_2)$
   - Cartesian products are the first step towards the `JOIN` operation
5. Cartesian Product and the Theta Join $\bowtie_\theta$
   - A **join** combines tuples from relation R_1, with tuples from another relation R_2 in some strucured way using a constraint
   - This constraint θ, determines how tuples from both relations will be combined

$$\bowtie_\theta = \sigma_\theta(R_1 \times R_2) \tag{9}$$

   - If θ consists of only equality constraints, this is called an equijoin. This is also known as an **inner join**
6. F, FOAF, FOAFOAF?
   - What if you wanted to identify all friend of a friend (of a friend) relationships,

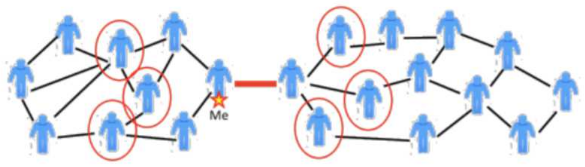


Figure 3: The first order FOAFs in the social network below are circled

   - To form this relation, take the cartesian product of R with itself. (Note: must rename one of the relations in order to rpoceed)
     - To denote this, let $\rho_X(E)$ be an operator the renames an operation from E to X
   - Now the above relation can be represented as:

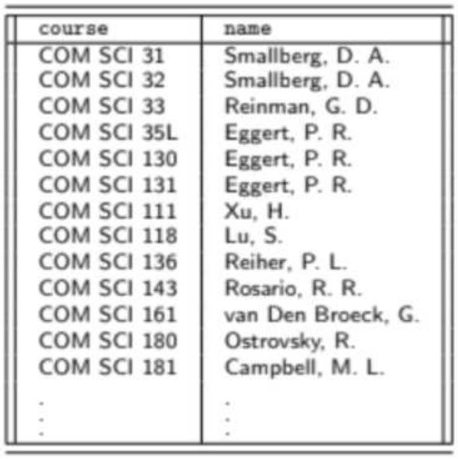
$$\Pi_{\text{L.user\_id,R.friend\_user\_id}}(\sigma_{\text{L.friend\_user\_id=R.user\_id}\wedge\text{L.user\_id}\neq\text{R.friend\_user\_id}}(\rho_L(R) \times R)) \tag{10}$$

7. Natural Join $\bowtie$
   - In relational algebra, the **natural join** allows us to join two relations together based on equality, using common attribute name

- The natural join simply joins allow common attributes
- This expression:

$$\Pi_{\text{course,name}}(instructor \bowtie teacher) \tag{11}$$

- Returns the following:



course	name
COM SCI 31	Smallberg, D. A.
COM SCI 32	Smallberg, D. A.
COM SCI 33	Reinman, G. D.
COM SCI 35L	Eggert, P. R.
COM SCI 130	Eggert, P. R.
COM SCI 131	Eggert, P. R.
COM SCI 111	Xu, H.
COM SCI 118	Lu, S.
COM SCI 136	Reiher, P. L.
COM SCI 143	Rosario, R. R.
COM SCI 161	van Den Broeck, G.
COM SCI 180	Ostrovsky, R.
COM SCI 181	Campbell, M. L.
⋮	⋮

- Unlike theta join, do not nead to alias attributes since the natural join knows which attributes to use as the join key
- If you try to join $R \bowtie S$, but R and S have no common attributes, this will just return the cartesian product $R \times S$
- Natural Joins are associative, so

$$(R \bowtie s) \bowtie T = R \bowtie (S \bowtie T) \tag{12}$$

8. Set Union $\cup$
    - The set union is a concatination of two sets of tuples that have the same structure
    - We can compute the set union between two relations, or between subsets of a single relation;
    - In the Youtube example, if we wanted to join both relations and extract the title, we could do something like:

```
SELECT title
FROM youtube_videos_us
UNION
SELECT title
FROM youtube_videos_gb
```

    - This query can also be represented as:

$$\Pi_{\text{title}}(R \cup S) = \Pi_{\text{title}}(R) \cup \Pi_{\text{title}}(S) \tag{13}$$

    - Note that both sets of tuples have the same schema, otherwise this is illegal
    - The union of these relations is one relation, containing all the tuples of the US location and all the tuples in the UK location
    - In general, performing a set union of tuples requires that the sets are compatible. This means that
        - $\|A\| = \|B\|$ (Same lengths)
        - $\forall a_i \in A, \forall b_i \in B, Dom(a_i) = Dom(b_i) \forall i$ (same domain)
    - Note that disjunctions can be written as

$$\Pi_g(\sigma_{\text{f}(\text{a}_1)\vee\ldots\vee\text{f}(\text{a}_\text{n})}(R)) = \Pi_g(\sigma_{\text{f}(\text{a}_1)}(R)) \cup \ldots \cup \Pi_g(\sigma_{\text{f}(\text{a}_\text{n})}(R)) \tag{14}$$

9. Set Difference -
    - The set difference operator takes in elements that are in the left relation but not the right relation

$$R - S = \{x \in R : x \notin S\} \tag{15}$$

- For example, if you want to retrieve only videos that have 500k - 1 million views, you can have the following expression:

$$\Pi_{\text{video\_id}}(\sigma_{\text{views}\leqslant 1000000}(youtube\_video)) - \Pi_{\text{video\_id}}(\sigma_{\text{views}<500000}(youtube\_video)) \tag{16}$$

- Which is equivalent to:

$$\Pi_{\text{video\_id}}(\sigma_{\text{views}\geqslant 500000 \wedge \text{views}\leqslant 1000000}(youtube\_video)) \tag{17}$$

10. Set Intersection $\cap$
    - The intersection of tuples is the subset of tuples that appears in both sets

$$S \cap T = \{x : x \in S, x \in T\} \tag{18}$$

    - Note that set intersection is redundant because

$$R \cap T = R - (R - S) \tag{19}$$

11. Rename ρ
    - Rename was introduced earlier, but note that you can also rename all its attributes at the same time

$$\rho_{\mathrm{X(A_1,A_2,\ldots,A_n)}}(E) \tag{20}$$

    - This statement says "rename relation E to X with attribute names given by $A_1, ..., A_n$"
12. Aggregation
    - Can represent aggregations on attributes using γ
    - Aggregations are functions applied to groups of tuples in a relation to summarize them
    - Examples include `SUM, AVG, MIN, MAX, DISTINCT-COUNT`
    - Example if you want average number of comment likes per video in youtube_comment

$$_{\text{video\_id}}\gamma_{\text{AVG(likes)}}(youtube\_comment) \tag{21}$$

    - Aggregates are specified as a subscript on the left of the operator
    - Formally, can aggregate multiple attributes simultaneosly and apply aggregation functions F_i to each attribute A_i

$$_{\mathrm{G_1,G_2,\ldots,G_n}}\gamma_{\mathrm{F_1(A_1),F_2(A_2),\ldots,F_n(A_n)}} \tag{22}$$

13. Relational Algebra vs SQL
    - One difference is that in relational algebra, we dealt with tuples, which will always be unique, this is not the case for SQL
    - A more realistic way is with multisets since multisets of tuples can contain duplicates

Lecture 3 - Introduction to SQL - 4/10/19

Why use PostgreSQL?

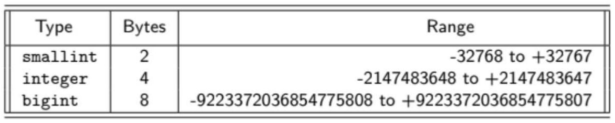
- MySQL is owned by Oracle, PostgreSQL is open-source
- PostgreSQL is largly ANSI SQL compliant
- PostgreSQL has better support for complex queries
- Users can read query execution plans
    - Lets you see what the database is going to do before you run the command
    - This is done with the command `EXPLAIN`
- PostgreSQL has appropriate security and authorization model
- Extensible and flexible type system similar to programming language
- Supports of indexing on JSON/XML

ANSI SQL Types

- ANSI SQL standard defines a lot of typess, supports different basic types such as
    - numeric
    - string/text
    - binary
    - dates and times

Number Types

- integer represents an integer defined in the database by the machine
- smallint represents a restricted domain int
- numeric(p, d) represents a fixed-point number containing p digits and d digits after the decimal
- real, double precision represents a double
- float(n) represents a floating-point number with n digits of precision
- In PostgreSQL, the only integer types supported are smallint, integer, and bigint
- These have the following characteristics



Type	Bytes	Range
`smallint`	2	-32768 to +32767
`integer`	4	-2147483648 to +2147483647
`bigint`	8	-9223372036854775808 to +9223372036854775807

Figure 1: Size and Range of Int Types in PostgreSQL

- PostgreSQL also supports Fixed-Point Types, for example `DECIMAL(p, d)` where p is the precision while d is the scale
    - d just means how many of the p digits come after of the decimal
- Inserting a value that has a scale larger than the scale declared in the schema will result in the number getting rounded
- Specific value of NaN exists, which is greater than all non-NaN values
- The maximum precision that can be specified is 1000
- **Note**: `numeric` and `decimal` are slow in PostgreSQL
- For Floating types, there are also the values `Infinity` and `-Infinity` as well as `NaN`

Strings

- char(n) is a fixed length character strong with a length of n characters
  - Note that this will get padded with 0s if you put in less than n chars
  - In PostgreSQL it actually pads with spaces
- varchar(n) is the variable length character string that can have up to n characters
- Note in PostgreSQL, casting a `varchar(m)` to `char(n)` where $m > n$, this will result in truncation
- Short strings ($len \leq 126$) have an overhead of one extra byte, longer strings have up to 4
- Unlike other database systems, there is no performance gain of using `character` over oher small string types.
  - `character` requires extra space for padding so for small strings it is the slowest

Dates and Times

- date is the calander date of the form (yyyy-mm-dd)
  - In PostgreSQL date is 4 bytes
- time is the time of day, of the form (HH-MM-SS)
  - In PostgreSQL, time is 8 bytes while timetz is 12 bytes
- timestamp - combines dat and time into the form (yyyy-mm-dd HH-MM-SS)
  - In PostgreSQL, timestamp and timestamptz are both 8 bytes, where timestamptz also stores the time zone
- epoch represents unix time
- `infinity` and `-infinity` which represents greater or less than all date times
- There are several SQL functions to get the current date or time: `CURRENT_DATE, CURRENT_TIME`, etc
- `allballs` is 00:00:00:00 UTC and only a valid `time` type

Note: The `NULL` value represents a lack of data

Binary Data

- The only binary data type is `bytea` which is used for storing raw bytes
- The boolean type also exists, which has 3 possible values (`TRUE`, `FALSE`, `NULL`)
- Bitstrings also exist in PostgreSQL which are arrays of `0` or `1`

ENUM Types

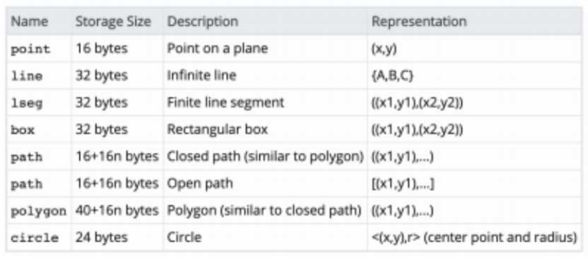
- MySQL is just a string represented internally as an integer. In PostgreSQL, it is an actual type.

```
CREATE TYPE mood AS ENUM ('sad', 'neutral', 'happy', 'mad')
CREATE TABLE person (
    name varchar(255),
    current_mood mood
)
```

- `ENUMS` are case sensitive and whitespace matters
- They take up 4 bytes
- Unlike MySQL, PostgreSQL ENUMS support type safety

Geometric Data Types

- PostgreSQL is actually really good for using geometric and spacial data types
  - MongoDB is also pretty good, but not relational like PostgreSQL
- These data types allow you to specify multiple shapes like lines, points, paths, etc



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)

Figure 2: PostgreSQL geometric data types

JSON and XML

- PostgreSQL allows native storage and retrieval of JSON and XML, you can for example index on a particular key that is nested in the JSON
- Once a table is assigned, it might be common to store json blobs as members of the column in a specific tuple. This allows you to specify an miscellaneous attributes

Creating a Relational Table

- Can use the `CREATE TABLE` syntax
- Example of creating a table

```
CREATE TYPE privacy_setting AS ENUM ('public ', 'private ', 'unlisted ');
CREATE TYPE location AS ( latitude point , longitude point );
CREATE TABLE youtube_video (
        video _id               character(11) UNIQUE NOT NULL,
        title                   varchar(100) NOT NULL,
        channel                 character(24) NOT NULL,
        cat_id                  smallint,
            -- could also be ENUM with names
        likes                   integer,
        dislikes                integer,
        views                   integer,
        post_date               timestamptz NOT NULL DEFAULT CURRENT_TIMESTAMP,
        duration                interval,
        privacy                 privacy_setting,
        content                 bytea NOT NULL,
        extra_data              jsonb,
        PRIMARY KEY (video_id)
        FOREIGN KEY (channel) REFERENCES youtube_channel(channel_id)
);
```

- Primary keys need to be `NOT NULL` and `UNIQUE`
- Foreign keys can be specified in a similar fashion to primary keys
- Can also add default values using `DEFAULT` followed by the value

Changing a Table's Schema

- If you need to change a table's schema, can do that with `ALTER TABLE`
- `ALTER TABLE` allows us to

    - Add and drop attributes
    - Add/drop primary and foreign keys
    - Rename relations and attributes
    - Add indices and constraints
    - Change data types of columns

- Here is an example of altering a table's schema

```
ALTER TABLE youtube_video
    DROP cat_id,
    MODIFY video_id CHAR(12),
        -- add a 12th character to video_id
    ADD flagged BIT AFTER channel,
        -- add a boolean flag for flagged videos
    ADD rowno INT FIRST,
    DROP PRIMARY KEY,
RENAME TO youtube_videos;
```

- Note: can also alter a table's foreign key using a similar syntax as above

Databases, Schemas, Relations/Tables

- Postgres Organizes data differently than MySQL. In MySQL databases are sets of tables where each database serves some different use case
- MySQL allows querying across your databases, Postgres does not
- In Postgres you want to create multiple schemas within one database
- Two common keywords in PostgreSQL are `CASCADE` and `RESTRICT`
- If you have a database where relation R with primary key K and a second relation S with that foreign key K, what happens when you delete R?
    - If you add `CASCADE`, S will be deleted as well, while `RESTRICT` will cause Postgres to see that S will be affected and throw an error.

Dropping, Deleting, and Truncating

- Can remove relations using `DROP`

```
DROP TABLE IF EXISTS youtube_videos;
```

- Can also drop Databases

```
DROP DATABASE IF EXISTS my_old_db;
```

- Dropping a schema can be done with:

```
DROP SCHEMA IF EXISTS my_schema;
```

- If you want to clear out all of the data in a table, but keep the relation and its schema, is to truncate the relation or delete from it

```
TRUNCATE youtube_videos
```

- If you want to delete specific data, can do a `DELETE FROM ... WHERE` clause

```
DELETE FROM youtube_videos WHERE views = 0;
```

Removing Duplicates

- In a standard SQL, `SELECT` removes duplicates because it is a projection (Π)
- If you want to know which professors are teaching some class, we can use `DISTINCT`

```sql
SELECT DISTINCT instructor
FROM instructor_course;
```

- `DISTINCT` is applied to all columns in the `SELECT`

Functions, Arithmetic and Renaming/Aliasing

- Can also do basic math on columns during a query

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

The ORDER Clause

- Sometimes the data we have returned from the `SELECT` clause needs to be ordered, this can be done with `ORDER BY`

```sql
SELECT
        uid,
        last_name,
        first_name
FROM roster
ORDER BY last_name, first_name;
```

- Note that the default sorting order is ascending order
- If you want descending order, can add a `DESC`
- `LIMIT` also lets you limit the number of results to s specified number

```sql
SELECT
        uid, last_name, first_name, gpa
FROM roster
ORDER BY
    gpa DESC, last_name, first_name
LIMIT 2;
```

Pagination

- If you have 200 students and you want to show 20 students per page, you can have a limit of 20, and then the next query will be offset by 20

```sql
SELECT
        uid,
        last_name,
        first_name
FROM roster
ORDER BY last_name, first_name
LIMIT 20
OFFSET 20;
```

Aggregation

- There are two types of aggregations, those over an entire relation and aggregations on a group
- For example, if you wanted summary statistics of GPA across the entire BruinBase, I can just issue a staight up query with aggregation functions:

```
SELECT
        AVG(gpa) AS average,
        MIN(gpa) AS minimum,
        MAX(gpa) AS maximum,
        COUNT(gpa) AS number_of_students
FROM BruinBase
```

- This will return one row consisting of all these statistics
- You can also have aggregations grouped by different attributes, for example if you wanted the average gpa by major

```
SELECT
        major,
        AVG(gpa)::numeric(3,2) AS average
FROM BruinBase
GROUP BY MAJOR
```

- You can then combine all these concepts into one giant query

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

- In general, the order of statements goes
  1. SELECT
  2. FROM
  3. JOIN
  4. WHERE
  5. GROUP BY
  6. HAVING
  7. ORDER BY
  8. LIMIT
  9. OFFSET

Lecture 4 - More on SQL - 4/15/19

HAVING

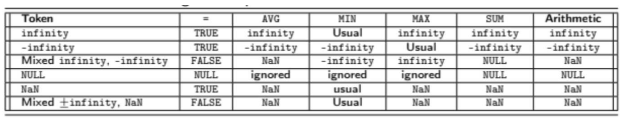
1. `WHERE` is essentially a pre-filter on the query, `HAVING` is a post-filter, where the `HAVING` clause is applied to the aggregation instead
2. Suppose you want the average GPA by major. Can specify major as a grouping variable and specify it in a `GROUP BY` and then using the `HAVING` clause to only keep the majors with GPAs over a 3.95

```
SELECT
        major
        AVG(gpa) AS average
FROM bruinbase
GROUP BY major
HAVING AVG(gpa) > 3.95
```

3. Note that average in the previous query is just for aesthetics, you cannot use aliases in `HAVING` clauses

Handling `NULL` Values

1. Note that `NULL = NULL` will return `NULL`, except comparisons involving the trivial `IS NULL`, `IS NOT NULL`, should always check for `NULL` like this
2. Aggregations ignore `NULL`
3. So here is what happens in PostgreSQL, roughly speaking, `NaN` has precendence, and `NULL` is ignored



Token	=	AVG	MIN	MAX	SUM	Arithmetic
infinity	TRUE	infinity	Usual	infinity	infinity	infinity
-infinity	TRUE	-infinity	-infinity	Usual	-infinity	-infinity
Mixed infinity, -infinity	FALSE	NaN	-infinity	infinity	NULL	NaN
NULL	NULL	ignored	ignored	ignored	NULL	NULL
NaN	TRUE	NaN	usual	NaN	NaN	NaN
Mixed ±infinity, NaN	FALSE	NaN	Usual	NaN	NaN	NaN

Figure 1: Arithmetic Characteristics

Processing Text with PostgreSQL

1. The `LIKE` Operator triest to match text with a pattern specified by %, which matches any number of characters wherever it is placed
   - % will match any number of characters, while _ will match 1 char
2. 'CAST' can be used to convert a particular value to a new data type for the current query only.
   - For example if you only want the `time` portion of a 'timestamp' you can cast:

```
SELECT CAST(NOW() AS DATE);
SELECT NOW()::date;
```

Control Flow

1. The `COALESCE()` operator returns the first non-NULL value in a collection
2. The `CASE` operator works similar to a switch in other languages
3. A `NULLIF()` construct returns `NULL` on invalid data, based on an equality
4. Suppose you want to take bruinbase and change gpa to midterm score to create a new table
   - If a student has a `NULL` midterm value, can force this `NULL` into a zero using `COALESCE`

```
SELECT
    uid, last, first
    COALESCE(midterm_score, 0) as midterm_score
FROM roster
```

- Can then combine all of these operations to create a cleaner view of the data
  ```
  SELECT
          uid,
          UPPER(CONCAT(last, ', ', first, COALESCE(' ' || middle, ''))) as name,
          CASE class_level
                  WHEN 'USR' THEN 'Senior'
                  WHEN 'UJR' THEN 'Junior'
                  WHEN 'USO' THEN 'Sophomore'
                  WHEN 'UFR' THEN 'Freshman'
          END AS class_level
          NULLIF(major, 'Undeclared') AS major,
          COALESCE(midterm_score, 0) as midterm_score
  FROM extended_roster
  ```

Modifying Data

Adding New Rows

1. `INSERT` can be used to add a new row into a table

   ```
   INSERT INTO relation VALUES ('val1', 'val2', ... , 'valn')
   ```

   Where (. . .) denotes a tuple of data, with strings in single quotes

2. If you only want to insert some of the values in the new row you must give the names of the columns

   ```
   INSERT INTO relation (col1, ... , colname) VALUES
       ('val1_1', ... , 'val1_n'), ('val2_1', ... , 'val2_n'),
       ...,
       ('valn_1', ... , 'valn_n');
   ```

3. If you don't specify all the columns in the `INSERT` statement, then the other columns will be set to their default values.

Modifying Rows

1. Rows in a table can be modified using the `UPDATE`. Which usually has one of the following forms

   ```
   UPDATE relation
   SET column = new_value
   WHERE some_condition
   ```

2. Can for example update all midterm scores for all students by adding 2 to all of them

   ```
   UPDATE extended_roster SET midterm_score = midterm_score + 2;
   ```

3. Can also pair an `UPDATE` with a `WHERE` clause if you want to only update a specific table

Joins: Querying Multiple Relations

1. `JOIN` is used for combining multiple relations into one during a query
2. `NATURAL JOIN` is typically very powerful but in relational algebra, but typically not used in SQL.
   - `NATURAL JOIN` just does not have a condition, it is picked automatically and is based on an equality
3. `INNER JOIN` is much more common because we explicitly state what we want to join on, so we don't have to rely on the relational database to find common attributes
4. **Note**: JOIN Types are usually explicitly, otherwise it defaults to an `INNER JOIN`

The Inner Join

1. The inner join is the most basic and common join, given to relations R and S, the inner join concatenates records where the join keys match on both relations, so it is $R \cap S$

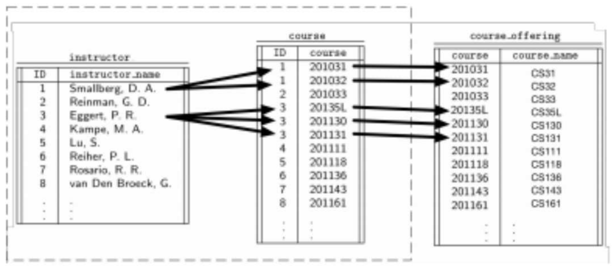
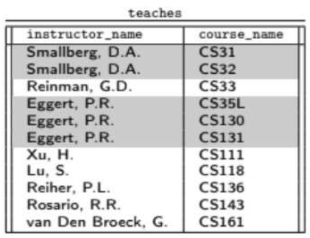


Figure 2: The relationship between course and course_offering is one to one

2. This above query will result in the following table



teaches

instructor_name	course_name
Smallberg, D.A.	CS31
Smallberg, D.A.	CS32
Reinman, G.D.	CS33
Eggert, P.R.	CS35L
Eggert, P.R.	CS130
Eggert, P.R.	CS131
Xu, H.	CS111
Lu, S.	CS118
Reiher, P.L.	CS136
Rosario, R.R.	CS143
van Den Broeck, G.	CS161

3. Note that if you have m rows and n copies of a row on the right, then the result will have mn combinations representing the particular set of values on the join keys.

Types of Joins

There are several types of joins, characterized by the following:

- Based on the operator used in the join clause
  - It is most common to use the = operator, however, it is possible to use all other boolean operators. These are called non-equijoin
- Based on how rows/records are matched and how we deal with rows/records
- Based on whether or not a second distinct table is used in the join
- Based on whether or not we explicitly state the columns used in the join.

The CROSS JOIN

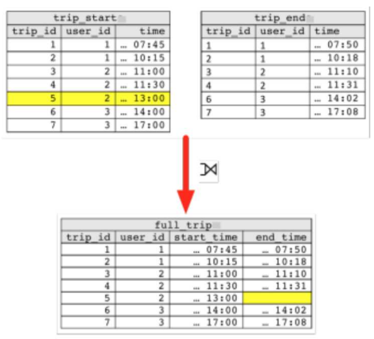
1. The cross join between two relations R and S is simply the Cartesian product denoted by $R \times S$
2. Cross joins are typically very rare and not used frequently

The OUTER JOIN

1. With `INNER JOIN`, you only return records where the values of the join key match both tables involved in the join. This is not the case with `OUTER JOINS`
2. `OUTER JOINS` will keep all entries and fill the columns from the other table as `NULL`
3. The `LEFT [OUTER] JOIN` will try to match records from the left table to the right table.

- If a match is found, we form a new record by concatenating the results
- If a match is not found, the values from the left table are kept and they are concatenated with `NULL` values from the right table

4. An example is if you have trips with a start time and an end time, but you don't have all the end times for any plethora of reasons, then a `LEFT OUTER JOIN` will do the following:



trip_start

trip_id	user_id	time
1	1	… 07:45
2	1	… 10:15
3	2	… 11:00
4	2	… 11:30
5	2	… 13:00
6	3	… 14:00
7	3	… 17:00

trip_end

trip_id	user_id	time
1	1	… 07:50
2	1	… 10:18
3	2	… 11:10
4	2	… 11:31
6	3	… 14:02
7	3	… 17:08

$\bowtie$

full_trip

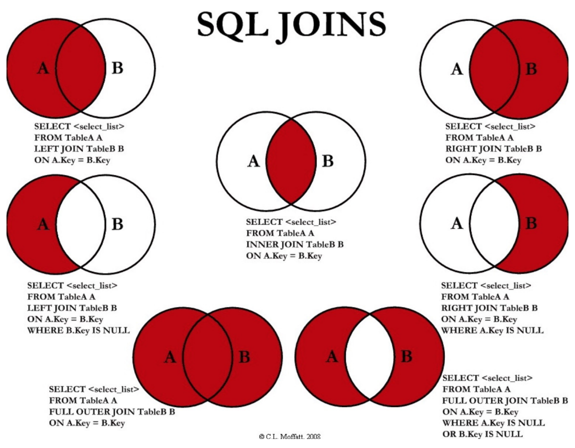
trip_id	user_id	start_time	end_time
1	1	… 07:45	… 07:50
2	1	… 10:15	… 10:18
3	2	… 11:00	… 11:10
4	2	… 11:30	… 11:31
5	2	… 13:00	
6	3	… 14:00	… 14:02
7	3	… 17:00	… 17:08

Figure 3: Notice how the 5th trip does not have a value, so the table returns null

5. The above query could be written as

```
SELECT
        L.trip _id AS trip_id,
        L.user _id AS user_id,
        L. time AS start_time,
        R. time AS end_time
FROM trip_start L
LEFT JOIN trip_end R
ON L.trip_id = R.trip_id AND L.user_id = R.user_id
```

6. The `RIGHT [OUTER] JOIN` does the exact opposite of the `LEFT [OUTER] JOIN` while the `FULL [OUTER] JOIN` combines the results of the left and right outer joins.



```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```

```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

Figure 4: Visualization of all the different types of joins

Lecture 5 - More on SQL - 4/15/19

Nested Queries

Nested Queries/Subqueries

- Subqueries are clauses that are run before having a WHERE clause query on that query.

```
SELECT
        major,
        average
FROM (
    SELECT
        major,
        AVG (gpa):: decimal (3,2) AS average
    FROM bruinbase
    GROUP BY major

) sq
WHERE average > 3.95
```

- Subqueries are always executed first, and in PostgreSQL, a subquery can be nested inside a SELECT, INSERT, UPDATE, DELETE, SET, or DO statement
- Usefull for constructing derived tables, set comparison, testing empty subsets/relations or testing uniqueness

Constructing Derived Tables

- The result of a subquery is sometimes called a derived table
- For example if you take a midterm scores table and want to compute the z-score of each student's midterm score by undergrad/grad. Can do this using the z score equation

$$z = \frac{x - \mu}{\sigma} \tag{1}$$

  Afer this, the corresponding query is:

```
SELECT
    uid , last , first , middle , L.career,
    midterm_score, (midterm_score - mean) / sd AS z_score
    FROM (
        SELECT
            career,
            AVG(midterm_score_) AS mean,
            STDDEV(midterm_score) AS sd
            extended_roster
        FROM roster
        GROUP BY Career
    )
    JOIN extended_roster R
    ON L.career = R.career
```

- Note that all derived tables must have an alias
- Can also use a subquery in the JOIN clause in the same way as the FROM

Dynamically Computing Constants: Correlated Subqueries

- Sometimes want to filter records by some boolean expression that requires a constant that cannot be precomputed
- Can use a subquery to compute the constant and then use it in the main query
- Say you want to identify students that scored at least a 0.5 standard deviations above the full class mean
  - Could do this by taking the previous query, drop the `GROUP BY`, throw it into a subquery, and then use a `WHERE` clause
  - Can precompute the $z = \frac{1}{2}\sigma + \mu$ in a subquery and use it into another query

    ```
    SELECT
        uid, last,
    FROM extended_roster
    WHERE midterm_score >
    ( -- this subquery makes the constant
      -- available to the outer query
        SELECT
            AVG (midterm_score + 0.5 * STDDEV (midterm_score))
        FROM extended_roster
    );
    ```

- **Note**: Correlated Subqueries are very inefficient. The subquery is recomputed for every row, wheras a standard subquery is executed only once.
- By using `EXPLAIN ANALYZE` before the query, we get an AST showing the query plan, the steps the RDBMS will take to carry out the query
- Scalar Subqueries compute a constant directly into a column via `SELECT`. For example:

  ```
  SELECT
      name,
      (SELECT max(pop) FROM cities WHERE
          cities.state = states.name)
  FROM states;
  ```

  Performance: Subqueries vs Joins

- Joins can be very expensive on very large tables. We typically want to filter before the join. Only keep the columns or rows we need to reduce the load. This filtering is usually done with a subquery
- Subqueries do not perform great on MySQL, and a `LEFT OUTER JOIN` is often preferable
- Joins, and subqueries before joins, tend to be the most performant. This always depends on the optimizer

Constaints

- We will dive into explicitly defining constraints on tables using the `CREATE TABLE` syntax
- Here are some examples of constraints and how we currently handle them
- In the SQL standard, can impose constraints on the table using the `CHECK` clause. The CHECK clause can be an expression or a subquery
- An example from hw1 is:

Constraint	How we handle it...
A unique identifier cannot be NULL	Mark as PRIMARY KEY or NOT NULL.
The UID of each student that lives on the hill must be present in the residents and dorms relation.	Use a foreign key.
The length of a Bird Scooter ride cannot be 0.	**Today**

Figure 1: List of current constraints

```sql
CREATE TABLE rides2017 (
    origin          char(4),
    destination     char(4),
    throughout      int,
    datetime        timestamp,
    PRIMARY KEY (origin, destination, datetime),
    CHECK(origin IN ('BALB', 'POWL', ...) AND destination IN ('BALB', 'POWL', ...))
);
```

- Can also put a subquery after the `IN` statement inside the check
- Using constraints is a judgement call. Some developers prefer to use a database just for storage and insert all logic into the application
- There are a few ways of specifying constraints when `CHECK` is not supported:
  1. Primary keys
  2. Foreign keys
  3. `UNIQUE`
  4. views
  5. triggers
  6. generated columns

UNIQUE

- Note that all values of a primary key must be unique, for a composite key, each pair must be unique across the table but the values in each column do not necessarily need to be unique
- `UNIQUE` works the same way as a primary key, the `UNIQUE` keyword can be written inline, or at the end of the `CREATE TABLE`
- If we want to specify that both the email address and the credit card number must form a unique pair can do this by:

```sql
CREATE TABLE user (
    user_id         SOME_TYPE_SPEC,
    ...
    ccnum           SOME_TYPE_SPEC,
    email           SOME_TYPE_SPEC
    UNIQUE(email, ccnum)
)
```

Views

Getting a Better VIEW

- So far, we have only worked with tables, which are implementations of relations, which are logic-level concepts. Views are now on the view-level which is what we would typically show to a user
- A view is essentially a shallow copy or a reference to a table
- A view is basically a "virtual table" that exposes only certain information to the user.
- Creating a view is very simple:

```
CREATE VIEW name_of_view AS
    (sql-query)
```

- What happens when the underlying data changes?
  - A standard view is just a window of the table, each time the view is accessed, the query for the view is updated
  - There is no performance gain to using a plain view over the table, but it does hide some data from the tables and it is always up-to-date

Materialized Views

- A **materialized view** caches a resultset on disk, we query it just like a table and we have a performance similar to a table
  - The query associated with materialized views may take a long time to generate the result, so materialized views must be refresed "manually" once in a while
- Example creation of a materialized view:

```
CREATE MATERIALIZED VIEW ucla_scooters AS
    SELECT
        scooter_id
        batter_power,
        last_known_location
    FROM scooter_status
    WHERE home_location - 1919;
```

- Just like variable references, views do not hold any data. Materialized views do cache data
- As the data in underlying tables change, a view recomputes the query on access, while a materialized view must be manually updated
- View definitions are stored in a schema

Updated Data Through Views

- One common characteristic used to descrive database systems is whether or not the database system allows users to modify the underlying tables
- The view may only use certain columns from the underlying tables, so if we try to insert a new row into a view, we must also insert the row into the underlying tables
- Each DBMS has different constraints on how to update views. In PostgreSQL

```
CREATE OR REPLACE VIEW UPDATE AS
... some query ...
```

Imposing Constraints with Views

- You can specify constraints on a table by using an updateable view that has `WITH CHECK OPTION`
- For example

  ```
  CREATE VIEW eligible_users AS
      SELECT *
      FROM user
      WHERE age >= 16
      WITH CHECK OPTION;
  ```

  The app now will insert a new user's information into the view instead of the table.

Triggers

- A trigger is a statement that the database executes automatically as a side-effect of modifying the database
- Triggers must satisfy two characteristics when created:
  1. **WHEN**: specify when the triffer is to be executed, and what conditions must be met for that to happen
  2. **WHAT**: Specify what the trigger will actually do.
- Triggers are created with the following syntax:

  ```
  CREATE TRIGGER <triggername>
  <Event>
  <Optional Referencing Clause>
  WHEN ( <optional expression or subquery> )
  <action>;
  ```

- The event spefifies the time point (`BEFORE` or `AFTER`) and a particular operation, like `INSERT`, `DELETE`, or `UPDATE`
- The condition is a boolean expression or subquery, while the referencing clause spedifies how we refer to the original row (for a `DELETE` or `UPDATE`)
  - The referencing clause looks like `REFERENCING OLD TABLE ROW AS <varname>`
- The action of a trigger is any SQL statement. An example `BEFORE INSERT` trigger is:

  ```
  CREATE TABLE account (acct_num INT , amount DECIMAL (10,2));
  CREATE TRIGGER ins_sum BEFORE INSERT ON account
      FOR EACH ROW SET @sum = @sum + NEW.amount;
  ```

  Which we can then trigger with:

  ```
  SET @sum = 0;
  INSERT INTO account VALUES(137, 14.98), (141,1937.50), (97,-100.00)
  SELECT @sum AS 'Total amount inserted'
  -- should return 1852.48
  ```

- In an `AFTER INSERT`, the condition will be checked, and then if it fails, the operation will be rolled back and undoed, it probably makes more sense to use a `BEFORE INSERT`

Lecture 6 - Authorization and Advanced SQL - 4/22/19

Authorization

1. A database consists of one or more users, where there is always one root user who is the administrator. The other users represent other processes that access your data
2. Database Management Systems have different permissions that users can have access for
3. Some of the most basic privileges are
   - Reading things
   - Creating thing
   - Inserting data
   - Changing things
   - Deleting things
4. Authorization is based on a Postgres user name (role). In Unix for example, a common group is called staff and everyone in staff has the same base permissions
5. A user'so privileges include all privileges granted to them and all privileges granted to his/her/the process roles
6. A template for gaining privileges:

   ```
   GRANT
       list_of_privileges
   ON
       database_table_schema_view name
   TO
       user_or_role_list;
       -- comma separated
   ```

7. An example of giving a user the ability to see data in the scooter_status views is:

   ```
   GRANT SELECT ON bird.ucla_scooter TO appuser;
   ```

8. Can also revoke privileges from users:

   ```
   REVOKE ALL PRIVILEGES ON bart.entry_exit FROM station;
   ```

9. Can also decide to **Pass the torch** where the user is assigned the privilege to grant privileges to others:

   ```
   GRANT ALL PRIVILEGES ON *.*
        TO user_name WITH GRANT OPTION;
   ```

10. If you want to undo the previous transaction, can use the `REVOKE GRANT OPTION FOR` syntax

    ```
    REVOKE GRANT OPTION FOR <privilege> ON bart
        FROM user_name;
    ```

11. Note that the authorization graph keeps track of which users gave permissions to whom. If a user U_i gave permission to U_j, then, if U_i has their privileges revoked, then so will U_j, but if U_j has recieved permissions from U_k, then it will keep its permissions
12. There are two actions we can take to revoke privileges from U_i:
    1. `CASCADE` revokes the privilege from U_i and all direct and indirect descendants,
    2. `RESTRICT` throws an error if any direct or indirect descendants have the authorization. So you have to recursively go through the leafs of the graph and revoke privileges first.

Applications

1. Connecting and using PostgresSQL consists of the following steps:
   - Connecting to the database using a driver
   - Constructing a query or prepared statement
   - Executing the query with a cursor
   - Iterating over the resultset using a cursor
   - Closing the connection
2. When formatting the SQL queries, it is always important to use string formatting in order to avoid SQL injection. So for python, use `s.format()` or `f"str"`
3. Notice that for example, we use the following piece of code:
   ```
   query = """SELECT id, student_last_name,
               student_first_name,
               student_middle_name, grade
               WHERE id=""" + request_args['student_id']
   ```
   What happens if the user enters something like: `1; DROP DATABASE students;`?
4. The query for this then becomes:
   ```
   SELECT id,
       student_last_name,
       student_first_name, student_middle_name,
       grade WHERE id=1; DROP DATABASE students;
   ```
   This is known as SQL injection.
5. SQL injection can also be used to steal private data.
   ```
   query = """SELECT
       uid,
       bol_username,
       bol_password
   FROM ucla_shibboleth
   WHERE bol_username='{}' AND ...""".format(user_input)
   ```
   Here the developer assumes that the user enters a valid username
6. But what if the user inputs `joebruin' OR 1=1; --`, leading to the code to become:
   ```
   SELECT
       uid,
       bol_username,
       bol_password
   FROM ucla_shibboleth
   WHERE bol_username='joebruin' OR 1=1; --' AND ...
   ```

SQL Injection and Prepared Statements

1. Instead, of using simple strings which are vulnerable to SQL injection, we should construct prepared statements which have two advantages:
   - They are faster when performing multiple queries of the same form.
   - Prevent SQL injection
2. Prepared statements are a way of creating query templates that can be used over and over again by simply creating a template, and then pop in the necessary values.

3. Here is an example query template that now lets me just pass in the proper data via api which is more efficient to code with:

   ```
   INSERT INTO table_name
        (Origin,Destination,DateTime,Throughput)
        VALUES (?,?,?,?)
            ON DUPLICATE KEY
            UPDATE Throughput = Throughput + 1;
   ```

4. Prepared statements prevent SQL injection by escaping user input, and treating it as a constant, so in the case of an attack the system would see:

   ```
   SELECT id,
        student_last_name,
        student_first_name,
        student_middle_name,
        grade
        WHERE id="1; DROP DATABASE students";
   ```

Putting it Together in Python

1. After connecting to the database, we can execute a query on a cursor. Then, the cursor provides traversal over the result
2. Note that it is important to close the connection when you are done since databases have a maximum number of concurrent users.
3. Here is an example of using the cursor in python:

   ```
   import psycopg2

   connection = psycopg2.connect(user="some_user",
       password="the_p@55w0rd",
       host="localhost",
       port="5432",
       database="my_database")
   cursor = connection.cursor()

   major = "MATH" # or some user input
   db_query = """

   SELECT uid, last, first FROM students WHERE major={};
   """.format(major)

   cursor.execute(query)

   # Can also use fetchone or fetchmany and iterate
   row = cursor.fetchall()
   for result in row:
       uid, last_name, first_name = result
       print("Student ID: {}, Name: {} {}".format(uid, first_name, last_name))
   conn.close()
   ```

Caching

1. Caching involves storing the results of past lookups (or important lookups) in a system that allows very fast retrieval.

2. Instead of querying a database server over and over each time we load the Bird app, the database server can cache scooter locations in a very fast key-value store or even on the phone's cache
3. It is however important to also update the cache, perhaps a good routine is to update every 30 min or so
4. PostgreSQL and other RDBMS implement caching as well, if you run two consecutive identical queries, you will notice that the second one finishes much faster than the first

Logging

1. When developing long-lasting applications, it's important to log as much as we can:
   - Login and logout, and authentication issues.
   - Impressions (views of content or pixels), interactions (clicks) and "successes" (conversions).
   - Visit path through the app or site.
   - Time spent on pages or screens.
   - Purchases
   - JavaScript and AJAX actions.
2. This information is often called metadata, but many times it can be just as important as actual payload data, since you can use it to view the state of the application and identify problems or bottlenecks
3. When logging, make sure to minimize private information in them, since if you get hacked you don't want that information leaked
4. Other things to do while logging to maintain privacy is:
   - Minimize the number of copies of private information in logs
   - Use a join key everywhere else, such as an integer or UUID
   - Hash super private data or just leave them out if they aren't needed

Review of SQL Functions

1. There are three ways to add new functions to SQL:
   1. Write a function with(out) arguments, with(out) a return value, as a SQL statement.
      - **Pro**: Easy to understand, and supports overloading.
      - **Con**: Slower than the other options, limited by declarative language
   2. Write a function in a procedural language like PL or pgSQL and compile it into a library. (dynamic loading)
      - **Pro**: Can be faster by allowing coder to specify how an algorithm is executed.
      - **Con**: Need to install (and perhaps distribute) object files. If the UDF interface in PostgreSQL changes, must be modified.
   3. Write an internal function that must be compiled into Postgres server. (static loading)
      - Pro: Fast!
      - Con: Have to modify Postgres server code.
2. Here is an example using the extended_roster table we used previously:

```
CREATE TABLE extended_roster (
    uid             varchar(9) PRIMARY KEY,
    last_name       varchar(50) NOT NULL,
    first_name      varchar(50) NOT NULL,
    middle_name     varchar(50),
    class_level     char(3),
    major           varchar(255) NOT NULL,
    career          char(1) NOT NULL,
    midterm_score   smallint
);
```

3. Now suppose we want to compute full names a lot using `CONCAT`, can put it in a function:

```
CREATE FUNCTION name(varchar, varchar, varchar) RETURNS varchar AS '
     SELECT CONCAT($1,'', '', $2, COALESCE('', '' || $3, ''''));
' LANGUAGE SQL;

-- To use
SELECT name(last_name, first_name, middle_name) FROM extended_roster;
```

4. Can also do a very similar thing for class levels:

```
CREATE FUNCTION classlevel(char) RETURNS varchar AS $$
    SELECT CASE ($1)
        WHEN 'USR' THEN 'Senior'
        WHEN 'UJR' THEN 'Junior'
        WHEN 'USO' THEN 'Sophomore'
        WHEN 'UFR' THEN 'Freshman'
        WHEN 'GD1' THEN 'PhD Student'
        WHEN 'GD2' THEN 'PhD Candidate'
        WHEN 'GMT' THEN 'Masters Student'
        WHEN 'CHA' THEN 'Chancellor Bossman'
        ELSE 'Unknown'
    END;
$$ LANGUAGE SQL;
```

Advanced SQL

Recursive Queries

1. Recursive queries allow us to extract recursive relationships, like all subordinates of a manager
2. The `WITH RECURSIVE` construct defines a recursive query. A recursive query has three parts:
   - The base set of results - the root of the search tree
   - Recursive term: one or more CTEs (Common Table Expressions) with a non-recursive term with `UNION` or `UNION ALL`
   - Termination check: when no more rows are returned.
3. An example is of the FOAF that was used a while back, find all subordinates of a manager with ID = 2:

```
CREATE TABLE employees (
    employee_id serial PRIMARY KEY,
    full_name varchar NOT NULL,
manager_id smallint
);
```

4. Can then run a query using:

```
WITH RECURSIVE subordinates AS (
    SELECT
        employee_id,
        manager_id,
        full_name
    FROM employees
    WHERE employee_id = 2
    UNION
    SELECT
        e.employee_id,
        e.manager_id,
        e.full_name
    FROM employees e
    INNER JOIN subordinates s ON s.employee_id = e.manager_id
)
SELECT employee_id, full_name FROM subordinates;
```

Window Functions

5. Window functions are similar to aggregation functions – but instead of collapsing a group, they take the order of the rows in the group into account.
6. These functions are used for:
   - Computing aggregates over rows in a group based on order;
   - Compute changes between rows in a group
7. For example, look at the chargebacks example from RIOT, previously a self-inner non-equijoin was used:

```
SELECT
    l.trans_id,
    l.customer_id,
    l.datetime,
SUM(r.result) AS prior_chargebacks
FROM transactions l
JOIN transactions r
ON l.customer_id = r.customer_id AND
    r.datetime <= l.datetime
GROUP BY l.customer_id, l.datetime, l.trans_id
ORDER BY l.trans_id;
```

8. This could instead be done by a window function: We could instead use a window function:

```
SELECT
    trans_id,
    customer_id,
    datetime,
    SUM(result) OVER (
        PARTITION BY customer_id
        ORDER BY datetime
    ) AS chargebacks
FROM transactions
ORDER BY trans_id;
```

Lecture 7 - OLTP and Data Warehousing - 4/22/19

OLTP

1. A relational database treats data as a 2-dimensional figure – a table. Rows correspond to records/tuples, and columns correspond to attributes.
2. Relational databases use seperation of concerns to divide data into individual tables. These tables must be joined together to exploit the relationships among data.
3. This separation of concerns reduces redundancy in data storage
4. The RDBMS optimizes for many random writes and reads.
5. Each one of these reads and writes is a result of a transaction, because of this, an RDBMS is an Online Transaction Processing System (OLTP)

OLAP (Online Analytical Processing)

1. OLAP, and Data Warehouses, are used for analytics only, and not for production systems. OLAP is multidimensional, rather than relational, but data may still look similar to a table to the user.
2. In OLAP, data is conceptualized as a hypercube to represent its multidimensional nature

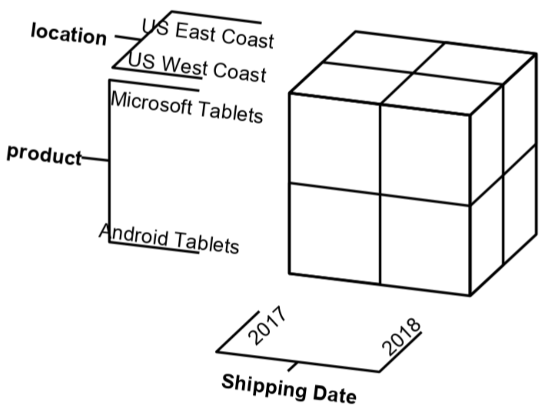


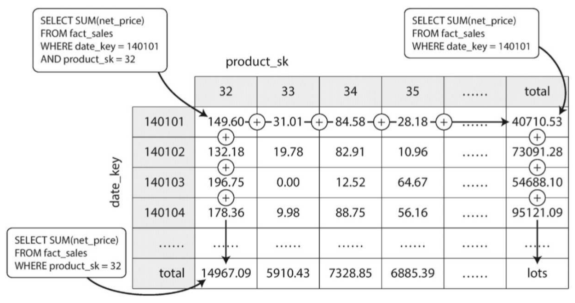
Figure 1: Amazon product sales data example representation

3. Another analogy for OLAP data is that they can be stored as tensors
4. OLAP is optimized for storing aggregates rather than raw data.
5. The user specifies two things to build an OLPA cube
   - The individual dimensions to analyze
   - aggregation functions to apply on each combination of dimensions.
6. Each time the data in the system is updated, the system computes all combinations of dimensions in the cube, offline, and applies the aggregation functions across each group.
7. The total number of combinations is given by

$$\sum_{k=1}^{n} \binom{n}{k} = 2^n - 1 \tag{1}$$

8. This is essentially like a `GROUP BY` but on every possible combination of columns

9. Typically data in OLAP is updated at a set schedule, usually overnight, since recomputing aggregations is very intensive process
10. OLAP makes heavy use of caching – aggregates across dimensions and their intersections are cached for quick access.
11. The main differences between OLAP and OLTP
    - OLAP cubes take a long time to process (very high latency);
    - Reads of aggregates are very fast (very low latency);
    - Data is likely to be out-of-date.
    - Typically append-only – no modifications or deletions
    - Read-only to users
    - For analytics, that's usually OK because we typically only look at full periods (i.e. full days).
12. During a down period, en ETL Job is executed against the OLTP/RDBMS and pulls the current state of the database, does some possible transformation to it, and then loads it into the data warehouse. The steps followed are:
    - **Extract**: Pull data from some source, usually an RDBMS.
    - **Transform**: Perform some transformation on the data, such as type casting, de-duplication, or processing JSON
    - **Load**: Write the raw data to the data warehouse. Under OLAP, aggregates will be computed.
13. Usually, an ETL job will run against a replicate of the RDBMS rather than the production database, since if you run the ETL on the already strained database, it will be very inefficient
14. Can also view the OLAP data as a 2d table:



```
SELECT SUM(net_price)
FROM fact_sales
WHERE date_key = 140101
AND product_sk = 32
```

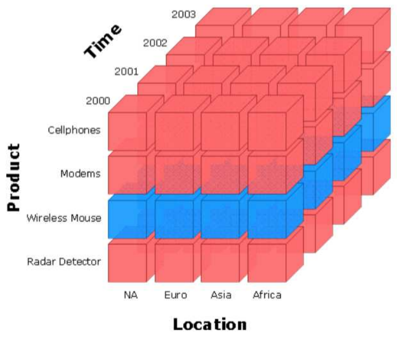
```
SELECT SUM(net_price)
FROM fact_sales
WHERE date_key = 140101
```

```
SELECT SUM(net_price)
FROM fact_sales
WHERE product_sk = 32
```

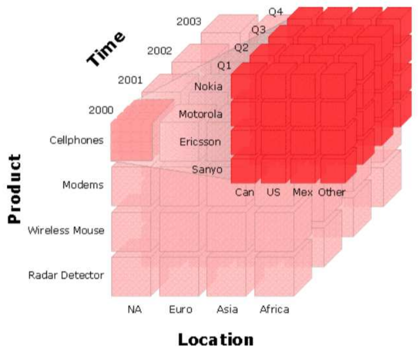
date_key \ product_sk	32	33	34	35	......	total
140101	149.60	31.01	84.58	28.18	......	40710.53
140102	132.18	19.78	82.91	10.96	......	73091.28
140103	196.75	0.00	12.52	64.67	......	54688.10
140104	178.36	9.98	88.75	56.16	......	95121.09
......	......	......	......	......	......	......
total	14967.09	5910.43	7328.85	6885.39	......	lots

Figure 2: 2D View of Data, here each entry is a different query

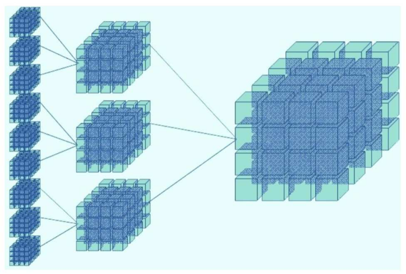
15. By pre-computing all the dimensions, there is some advanced OLAP operations that we can perform easily:
    - **Slicing**: selects one particular dimension from a cube and returns the sub-cube



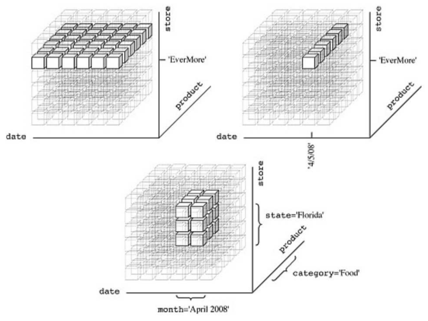
- **Dicing**: slicing for multiple dimensions



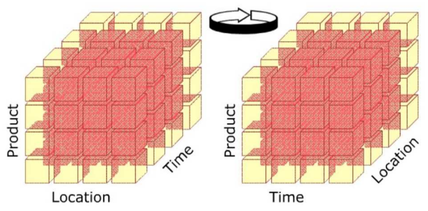
- **Rollup**: creates aggregates on dimensions, including up a hierarchy,



- **Drill Down**: is the opposite of rollup - given an aggregates we can display more fine aggregates for more specific dimensions



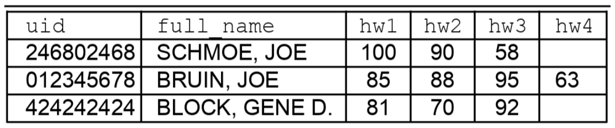
- **Pivot**: rotates data between two different formats: long and wide



16. Note that OLAP is a set of operations we can do on that data
17. Many data warehouses now use a column-oriented model rather than relational or OLAP cube. Since we want something that can across across a column quickly, rather than scanning across rows quickly.
18. Data warehouse systems are often denormalized to allow quick access to important information.

Two Common Data Formats: Long and Wide

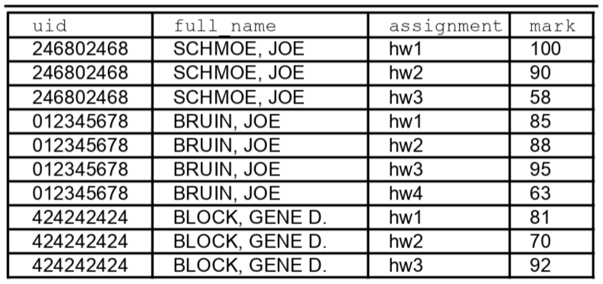
1. Suppose we have a homework_grades table: each student has multiple grades, one for each homework assignment.
2. The first format is wide; it involves multiple columns, one per homework in this case, and one row per student.



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

Figure 3: In this case, wide is the more natural of the 2

3. The second format is long; it involves multiple rows per unit. In the case of homework_grades, multiple rows per student.



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

Figure 4: Here, each row is the hw assignment

4. The long format is not the most intuitive, but it has several advantages
   - We do not need to know a priori how many homework assignments there will be
   - Assigning another homework is as simple as inserting a new row for each student.
   - Removes redundancy, there is no `NULL` for students who didn't turn in their homework
5. For storage, Long format is more preferable, since it provides better flexibility, and efficiency (Faster to use more rows than more columns)
6. Converting between long and wide format, particularly long to wide, is called pivoting.

Pivoting

1. OLAP systems and Data Warehouses support pivoting natively with the PIVOT keyword. RDBMS do not, PostgreSQL exposes it via an extension called tablefunc with the crosstab
2. Here is an example of convering the homework_grades table to a wide from long:

```sql
SELECT
    uid, full_name,
    SUM(CASE assignment WHEN 'hw1' THEN mark ELSE 0 END) AS hw1,
    SUM(CASE assignment WHEN 'hw2' THEN mark ELSE 0 END) AS hw2,
    SUM(CASE assignment WHEN 'hw3' THEN mark ELSE 0 END) AS hw3,
    SUM(CASE assignment WHEN 'hw4' THEN mark ELSE 0 END) AS hw4
FROM homework_grades
GROUP BY uid, full_name;
```

   There are 4 `CASE` statements, each one generates a new

Rollup Tables

1. We have used `GROUP BY` extensively, a rollup is more sophisticated, it provides aggregates across all levels of the heirarchy. An example for the hw table is:
   - Average midterm score by TA, section, by number of lectures attended;
   - Average midterm score by TA and section; average midterm score by TA;
   - Average midterm score for entire lecture
2. In PostgreSQL, we can add a ROLLUP to a GROUP BY that allows us to compute aggregations, and then adds another row

```
SELECT
    ta,
    AVG(points) as ta_average
FROM midterm_grades
GROUP BY ROLLUP(ta);
```

Overview of OLAP

1. Series of data operations designed for fast analytics
2. Data is multidimensional and dimensions and their interactions are pre-computed
3. Use some redundancy to improve querying times for fast analytics.
4. Difference between OLAP and OLTP is that OLAP is for fast analytics while OLTP is designed for short transactional writes and updates.

Lecture 8 - MapReduce - 4/29/19

MapReduce

- MapReduce consists of two main dataflow stages, but there are actually several more:
  1. Input
  2. MAP
  3. Partition
  4. Comparison
  5. REDUCE
  6. Output
- Suppose we have the text from every English webpage in a cluster, then input here is each webpage as a single HTML file.
  - By default, Hadoop reads lines from files, where each line is a record. This is often not appropriate, for example a Wikipedia dump is millions of lines in an XML file, thats why we need a special class called InputFormat
- InputFormat specifies how records are laid out in files and how to divide the records into splits.
- Splits are used to define parallelism; if we split 1 million records into groups of 1000, we can process the data in 1000 batches concurrently. If we have 1000 CPUs, we can finish the job 1000x faster
- Back to the example at hand, we break down the problem to each webpage. We will do some text processing like removing html. If we try to scale this using bash with a command like:

  ```
  cat file | awk ... | sort | uniq -c
  ```

  this will create thousands of subprocesses, and those don't share memory so we will run out of space in memory
- Method 1: (does not require extra memory)
  - We don't need any more memory if we simply output a count each time we encounter a word, even if words appear more than once:
  - For the sentence:

  ```
  The quick brown fox jumps over the lazy dog
  ```

  we would output

Word	Count
The	1
quick	1
brown	1
fox	1
jumps	1
over	1
the	1
lazy	1
dog	1

  This is called the mapping phase
- Method 2: Extra Memory Required (map and combine)
  - We can use extra memory by using our original method, using a data structure to aggregate counts

from one file and then output the contents of the data structure to disk.

– So in the same previous example, we would create a table:

Word	Count
quick	1
brown	1
fox	1
jumps	1
over	1
the	2
lazy	1
dog	1

- During the Map Phase, The data are then partitioned. In simple cases, this is a simple sort by key (word), but we can also sort by the value of some hash function.

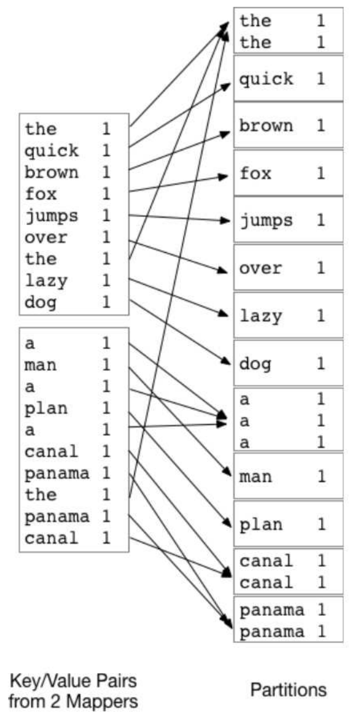


Figure 1: A visual description of the map phase

The Reduce Phase

- Each reduce task on a reducer receives a group of key-value pairs, all with the same key, and they may come from several machines, so a merge and sort is performed a final time.
- This however causes problems where some words are much more likely to appear and thus will have a key skew, causing some keys to take hours to compute
- The solution to this problem is to create a subkey, so that records are more evenly shuffled to reducers.

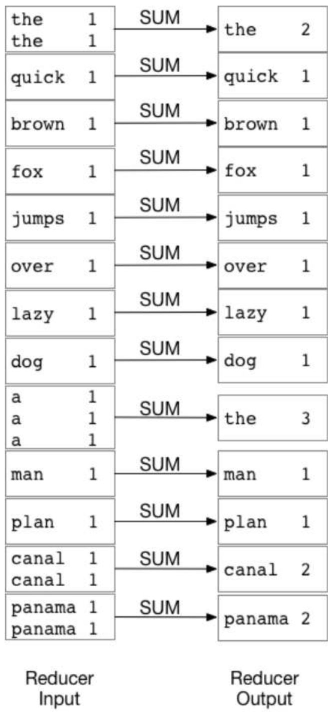


Figure 2: An example of what the reduce phase looks like

- We can use an OutputFormat (the output) phase if we want to use some other format like Parquet instead of the default one record per file

MapReduce: The Big Picture

- MapReduce functions on each split independently. MapReduce is a shared-nothing model, and is also embarrassingly parallel

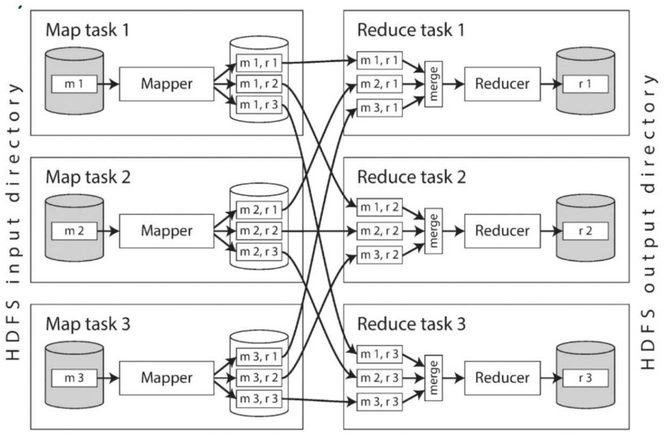


Figure 3: MapReduce: The Big Picture

- It is very obvious how to divide the problem to make it parallel without any horizontal dependencies among splits.

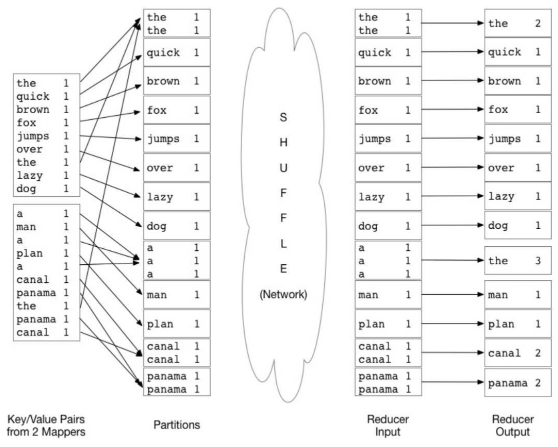


Figure 4: Diagram of Our Data

- Here is a python example of MapReduce
  - **Python Mapper**

```
#!/usr/bin/env python
"""mapper.py"""
import sys

# input comes from STDIN (standard input)
# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output). What we output here
        # will be the input for the Reduce step, i.e. the input for reducer.py
        # tab-delimited; the trivial word count is 1
        print("\{\}\t\{\}".format(word, str(1)))
```

  - **Python Reducer**

```
#!/usr/bin/env python
""" reducer.py """
from operator import itemgetter, sys

current_word = None; current_count = 0; word = None

for line in sys.stdin:
    line = line.strip()
    # split the key/value pair from the mapper
    word, count = line.split('\t', 1)
    count = int(count)
    # this IF only works because Hadoop sorts map output by key (word)
    # before it is passed to the reducer
    if current_word == word:
            current_count += count
        else:
            if current_word:
                print("{}\t{}".format(current_word, str(current_count))
            current_count = count
            current_word = word
    # Do not forget to output the last word if needed!
    if current_word == word:
        print("{}\t{}".format(current_word, str(current_count))
```

SQL in MapReduce

- MapReduce is typically used on data that is not already in a relational format.
- SQL operations can be performed using combinations of map and reduce phases, and even the absence of them:
  - A SQL `WHERE` clause or σ can be represented as a map without a reduce phase
  - An aggregation γ can be represented as both a map that constructs key-value pairs with groups and a reduce phase that computes the aggregate function on each group
- Queries with multiple operations can chain together several map and reduce stages to answer the query.
- Unlike in RDBMS, when using a subquery within a JOIN it is better to materialize the intermediate result into a temporary table
  - This is because a subquery may execute concurrently with a join operation, and a join will recompute records as more records are created in the subquery

Beyond MapReduce

- Relational algebra operators and trees of operators can be represented as a series of maps and reduce steps, but this can be cumbersome
- For example, a join in relational algebra is one step but in MapReduce, a join is two operations
- More recent parallel systems have added support for one operation joins.
- Most data workflows comprise a series of related steps called a pipeline
- We want to think of our pipeline as consisting of a series of algebraic steps that take arbitrary data as input, and output other arbitrary data depending on the context, just like RDBMS.
- Apache Spark is an example of such a model, where it uses Resilient Distributed Datasets (RDDs) as input and output to/from operators instead of relations.

Apache Spark

- An RDD is resilient because when one node in the distributed system fails, there is a replicate elsewhere in system
- The beauty of spark is that its distributed nature is transparent to the user, with data treated as a single data structure as if it was on a local machine
- Some noticable advantages of Spark over Hadoop is:
  - Computations are stored in RAM, and stay there until the user specifically requests to persist them to disk.
    - Calling collect() will persist results to disk.
  - As operations are entered by the user, Spark constructs an execution DAG
  - Spark can more easily handle iterative problems such as those in machine learning.
  - Spark has been measured to be up to 100x faster than Hadoop when using RAM, and 10x faster when using disk/HDFS.

Lecture 9 - Data Storage and Indices - 5/6/19

Data Storage

Various Methods of Data Storage

- There are variety of ways we can store data on a system, the one we pick will depend on the speed of access, cost per unit, and data reliability
- **Cache**
  - fast, but expensive
  - CPUs, GPUs and hard disks all have caches
  - cache stores frequently accessed values so that we don't have to reaccess the RDBMS
- **Main Memory/RAM**
  - Fast and expensive
  - Can serve as cache
  - Data doesn't usually fit into RAM (Spark rarely relies on RAM), need fast way to swapp between disk and RAM
  - Volatile memory
- **Solid State**
  - Hybrid of RAM and hard disk
  - SSDs offer fast seek and random access almost as fast as RAM, but the storage is nonvolatile
  - SSDs commonly used for RDBMS
- **Magnetic (Hard) Disks:**
  - Very cheap but very slow
  - Used for long-term storage
  - Hard disks up to 10-15 TB in 2019
  - Prone to nasty failures
- **Optical Media**
  - No longer very common (things like DVD(4.7-8.5GB), BluRay (25-128GB))
- **Tape Storage**
  - very, very cheap and just as slow
  - great for long term backup
  - requires sequential access
- **Cloud Storage**
  - Typically is one of the previously mentioned types
  - Lots of tradeoffs
  - Slow

Anatomy of Hard Disk

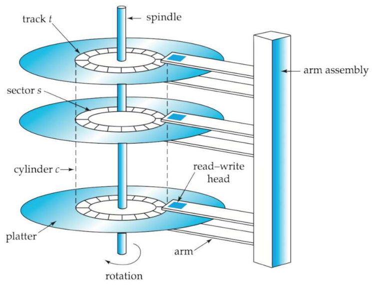


Figure 1: Hard Disk Architecture

- A hard disk consists of a series of circular platters.
- Each platter is divided into circular tracks, with each track consisting of many sectors (semi-circles).

- If we recognize the same track of all platters, we have a cylinder
- A motor rotates platters upon a spindle with an arm assembly
- The read/write head floats just above the disk platter surface.
- When the head touches the disk, it can scratch surface causing a head crash.
- Hard disks over some error correction by remapping bad sectors.
- When a hard disk controller detects a secotr of disk that is corrupted, a note will be made in the disk's nonvolatile memory to map all requests to that sector to some other sector
- Hard disk performance can be measured in terms of access time, seek time, rotational latency, data transfer rate, and mean time between failures
  - Note that in this case, access time refers to seek time + latency
- Logical Data Access on Hard Disks
  - **Sequential**: Successive disk requests are for successive block numbers.
  - **Random Access**: Successive requests are for random blocks located throughout the disk.
- Disks are very slow, to speed them up, several techniques have been developed
  - **Scheduling**: When a request is made, requests are ordered by cylinder and then read in order of cylinder to minimized disk head movement
  - **File Organization**: We can arrange disk blocks in the order we expect them to be requested.
  - **Nonvolatile write buffers:**: When writing blocks, may have to wait for the write head to be ready, so data is queued somewhere to wait, but if there is a power failure, data must not be lost so write requests first write to NVRAM then when ready, they are written to disk

Solid State Drives

- SSDs on the other hand are optimized for random accesses, so sequential accesses just boil down to random accesses
- writes to flash and SSDs are complicated since once a block has been written it cannot be easily overwritten
- Performance on SSDs is measured as follows:
  1. Number of block reads per second. Reads can occur in parallel.
  2. Number of block writes per second.
  3. Data transfer rate for sequential reads and writes.

Data Architecture in RDBMS

- A database maps logical data to files that are physically stored on the hard disk.
- A file consists of a sequence of records, onto disk blocks.
- Databases work with records that are stored in files.
- For efficiency, as well as to support data recovery databases must continue to be aware of blocks

Organizing Records of Files

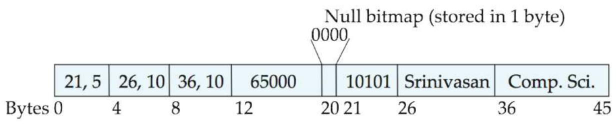
- Tuples (records) from distinct relations will be of different sizes, so there are two ways to store records into files:

  1. Assume all records in the same file have the same length
  2. Allow records in the same file to differ in length

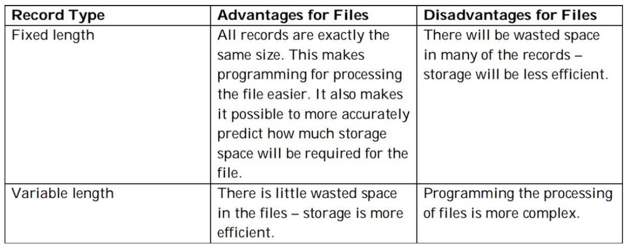
- Problems with fixed-sized records

  - A set of fixed records may not be a multiple of the block size and so there is no disk access benefit out of it because records cross block boundaries, so certain records requires at least 2 accesses to read or write. This is not efficient
  - Difficult to delete a record because then there will be a gap between records. This can lead to fragmentation.

- Solutions to fragmentation
  1. Fill the gap with subsequent records, but this will take time to fill
  2. Introduce a file header that contains metadata about the file. Cache the addresses of the deleted records and use it to seek the proper block for inserting a new record
- One solution to storing variable length records: Split record into 4 parts
  1. A table of contents containing byte offsets and lengths for the variable length fields;
  2. A fixed part with fixed data types
  3. A bitmask denoting which attributes are null
  4. A variable part for variable-length attributes that is indexed using the "table of contents."
- An example of a variable length record may look like:



- Each Block contains a header with the number of records in the block, the location of end of free space in the block, and the location and size of each record
- The records themselves are stored in some sequential order within a block
- Al references to records, should reference the header of the block



Record Type	Advantages for Files	Disadvantages for Files
Fixed length	All records are exactly the same size. This makes programming for processing the file easier. It also makes it possible to more accurately predict how much storage space will be required for the file.	There will be wasted space in many of the records – storage will be less efficient.
Variable length	There is little wasted space in the files – storage is more efficient.	Programming the processing of files is more complex.

Figure 2: Fixed vs Variable Length

- There are three main ways of organizing records into files
  - **Heap**: Records are put into any block that has room, usually at the end of the file. Once stored the record doesn't move. Good for for batch insert
  - **Sequential**: Records are stored in sorted order by some search key. becomes inefficient after many modifications since these operations are $\mathcal{O}(pnq)$
  - **B+tree**: Similar to sequential storage, but more efficient after many modifications. Operations about $\mathcal{O}(n \log n)$
  - **Multitable Clustering**: Multiple tables are stored in one file, and even in the same block. This can improve join performance.
  - **Hashing**: Records are stored in a particular block of the file based on the hash value mapping to a block number.
- With sequential organization, records are stored in a file in some sorted order by key.
- When we delete records, we modify a pointer chain just like when deleting from a linked list.

Database Buffer

- From an OS, main memory contains a buffer that stores copies of disk blocks, and this is managed by the buffer manager, essentially a virtual memory manager.
- The buffer manager does the following:

  - Figuring out which blocks must be evicted from the buffer when the buffer fills up. This is done with an LRU cache where the list recently used block is evicted
  - Figuring out which blocks must be evicted from the buffer when the buffer fills up, like don't write to disk when block is being updated
  - Forcing blocks back to disk even if there is room in the buffer
- Databases use a toss immediately schema: as soon as the final row has been processed, the blocks associated with the queried tables are cleared

Indexing and Ordered Indices on Sequential Files

- Usually when write a query we only want to search on particular attributes including primary keys and perform joins on foreign keys or other join keys. A naive database implementation would rely on $\mathcal{O}(pnq)$ full table scan to find matches for `JOIN`, `WHERE`, etc. What data structures allow for a faster lookup?
- There are two types of indexes:
  1. **Ordered indices**: Based on a sorting of the sort-key values and records
  2. **Hash indices**: Based on a uniform distribution of values across a range of buckets.
- The type of index used depends on several factors:
  1. **Acess Type**: equality or range search?
  2. **Access Time**: time it 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.

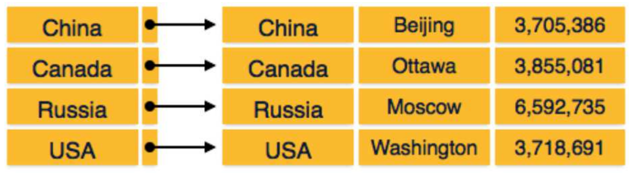
Ordered Indices

- In an ordered index, the tuples in a relation are sorted by the key and distributed to blocks, because keys are stored sequentially in order, you can use binary search to find them in $\mathcal{O}(logkp)$ where k is the number of search keys

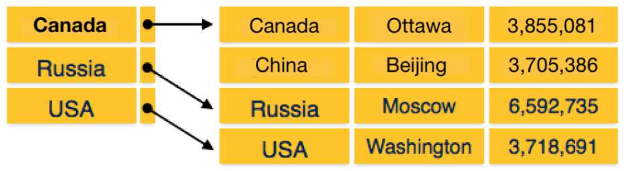
- In a primary/clustered index, there is a unique search key, usually the primary key, and data in the data files is sorted by this key.

- The primary index tends to be pretty fast. When you lookup a key you get back a block number and records offset

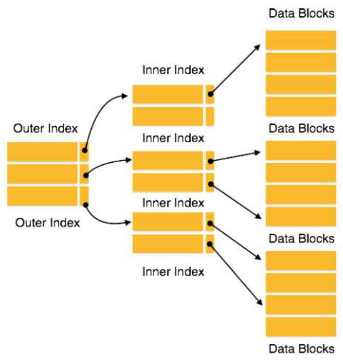
- Primary indices are usually implemented as dense indexes, meaning that there is an entry in the index for every value of the search-key. The index takes less space than the record, so it requires fewer blocks. If the index fits in RAM, only one I/O is needed to retrieve the record from disk. This is known as index sequential



- We can also have a primary sparse index, which just means that not every value of the search key is in the index. When searching for an index, will find the nearest key to the one we want



- If the index is sorted by key amd the data file is sorted by something else, the index is called secondary/non-clustedered index. (key doesn't have to be unique)
- May also use a multilevel approach if we have something like a composite key



- What happens when we try to insert a record and the disk block where it should be stored is already full? Normally, it would require a chain of overflow blocks and this becomes painful, so instead a B+ tree is used

Ordered Indices and B+Trees

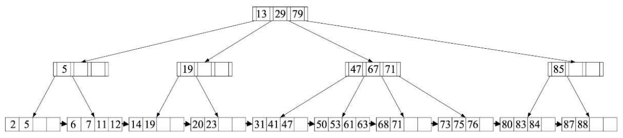
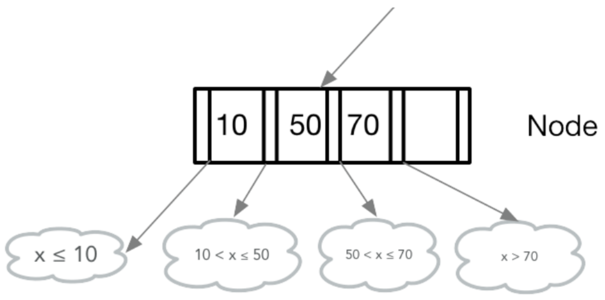


Figure 3: A B+ tree looks like the following

- A B+ tree has the following characteristics:
  1. Is a self-balancing tree N-ary tree
  2. Requires that paths from root to every leaf have the same length
  3. Contains a root, internal nodes, and leaves.
  4. The root is either a leaf, or has 2 or more children.
  5. High fanout to reduce the number of block reads
  6. Represents one block read at each level of the tree
- B trees generally store both keys and data in internal and leaf nodes. Frequently accessed records in an index can be stored closer to the root in a B-tree, which is not possible in a B+ tree.
- For a given internal node, the branching factor n indicates the capacity (in pointers) of the node and m represents the actual number of child nodes it has.
- The root represents the full domain of keys in the tree and each internal node is a subinterval of that domain.
- If k is less than all of the keys in the node, we traverse to the far left. If k is greater than all of the keys in the node, we traverse to the far right.



- B+ trees allow for fast lookups but there is performance overhead on insertion and deletion, and of course space overhead.
- Each time we traverse a level in the B+ tree, we perform disk I/O – we read an entire block.
- If there are n items per node, and K keys, the search complexity is $\mathcal{O}(\log_n(2)\log_{\lceil \frac{n}{2} \rceil}(K))$
- Inserting into the B+ Tree follows the following recipe:
  1. Find the position in the tree where the key should go.
  2. If that node is not full, insert the key. Else:
     1. Split the node into two pieces. The original keeps half of the keys. The new node keeys the remaining half.
     2. Move the middle key to the parent and insert a new in the parent to the new node.
     3. Keep doing this until there are no more overflows in the parent.
     4. If the root splits, treat it as if it has an empty parent and split.
- Some disadvantages of B+ trees
  - Best performance when there are no duplicate keys.
  - Variable length attributes, like strings, make it diffcult to store in a tree under this treatment
  - Bulk loading arbitrary data into a B+ tree is very inefficient.

Lecture 10 - Hashing and Other Indices- 5/8/19

Note: Can create an index in SQL with the following syntax:

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

Hash Indices

- Hash indices are widely used for building indices in main memory/RAM but can also live on disks.
- They can permanently reside in RAM or on disk (for fast lookups), or be computed on the fly during a join.
- Hash indices are great at answering equality queries, not range queries
- We define a bucket as a container that houses one or more records. The bucket could link records together using a linked list of index entries (multilevel) or one of records. We will let B be the set of buckets.
- If the index is on disk, a bucket would be linked list of blocks
- Let K be the set of all search-key values. and let h be a hash function $h : K \to B$. That is, h maps a key K_i to a particular bucket b.
- To insert a new record into the index, we compute the hash function on the key $h(k_i)$ which gives a bucket address for the record. Then we add an index entry for the record to the list at offset i.
- To perform a lookup on k_i, we compute $h(k_i)$ then iterate through the bucket to process the records.
- It is possible that $\exists i, j, i \neq j : h(k_i) = h(k_j)$
- So the bucket $h(k_i)$ contains records with the key k_i and k_j so we have to search for records containing the proper key
- Deletions can be done by computing $h(k_i)$ and find all the records that have k_i and delete them from the linked list
- If the hash index is on disk, during insertion is done by locating the bucket and if there is space in the bucket, the record is stored, otherwise there is a bucket overflow so an overflow bucket is constructed for that record and subsequent records
- Can't store everything in one bucket because of block I/O, which means we want efficient storing of records
- Bucket overflow
  - **Insufficient Buckets**: for the number of records we want to index
  - **Bucket Skew**: Some buckets contain a lot of records and a lot of overflow buckets, whereas the rest do not have any overflow buckets. This is caused by non-uniform hash function or imbalanced data
- We want to minimize the probability of bucket overflow. We generally choose $\frac{n_r}{f}(1+d)$ where n_r is the number of records, f is the number of records per bucket and d is a fudge faster (usually 0.2 so 20% of buckets are empty)
- We may waste some space, but we get quicker performance because we won't have to iterate a linked list of overflow buckets.
- When we allocate the number of buckets a priori, this is called **static hashing**.
- To avoid bucket overflows, can do one of the following
  1. Choose a hash function based on the current file size. This option will result in performance degradation as the database grows.
  2. Choose a hash function based on the anticipated size of the file at some point in the future. Although performance degradation is avoided, a significant amount of space may be wasted initially.

3. Periodically reorganize the hash structure in response to file growth: new hash function, recomputing hash values on every record, generating new bucket assignments. **THIS IS SLOW**

- Instead, we can grow or shrink the number of buckets we use incrementally, called dynamic hashing
  - linear hashing
  - extendible hashing

Extendible Hashing

- Under extendible hashing, the buckets replicate and coalesce as the database grows and shrinks.
- This allows for space efficiency while retaining performance* This allows for space efficiency while retaining performance
- Under extendible hashing, we use a hash function that is random and uniform, but generates values over a very large range
- Buckets are not preallocated, they are allocated on the fly. At any time, we only use i of the b bits, where $i \leq b$ initially. We treat this as a prefix for any future higher order buckets that are created. i changes as the database grows and shrinks.

Indexing Other Content Types

- Sequential indexing, B+ trees and hash indices are the most common indexing methods.
- The other realistic data types you may encounter are spatial, temporal, and streaming
- You can index spatial data which will construct a system that lets you extract points that are within a certain distance of the user, usually this is a circle and is called a **nearest neighbor query**
- In the 2-3D plane, we use a special tree called a k-d tree where each boundary line represents one node in a k-d tree
- To index temporal data, a lot of it is borrowing over from spacial data in 1D

Straming Data

- Many times we only want to know if a particular key exists in a set, it without retrieving the records. If the set is extremely large and does not fit in RAM and would take forever to distribute in a network.
- If we have to stream the data across the slow network, faster throughput can be acheived by reducing space complexity
- Given some key k, we can query the Bloom Filter to determine whether or not k is in the set. The Bloom Filter may respond yes or no.
- We can trust an answer of no from the Bloom Filter. That is, there are no false negatives.
- A Bloom Filter is a bit array of m bits, initialized to 0. We have a set of hash functions F where each hash function $f \in F$ hashes each input k to one of the m bits. Assuming $f(k)$ is uniformly distributed. Usually the number of elements in F is way less than m
- To insert into the filter, computer $f(k)$ for all $f \in F$ to get a series of $|F|$ bit positions. We set each of these bits to 1. This is $\mathcal{O}(|F|)$
- Deleting is not possible because it may lead to conflicts with other search keys which violates the perfect false negative
- A Bloom Filter is a class of data structures called probabilistic data structures. They do not always yield precise responses to queries, but are very important to streaming systems without access to past data

Lecture 11 - Query Processing and Optimization 5/13/19

Query Processing

- Query processing takes a SQL query and converts it into a higher order language understood by the RDBMS (Relational Algebra)
- Query Processing results in expressions that can be used at the physical level of the file system, and a variety of optimizations
- First the parser checks the syntax of the query and verfies all the relation names
- The system then constructs a parse-tree of the query which translates to a relational algebra expression
- Several different queries may lead to the exact same resultset, and one query can also be represented by multiple relational algebra expressions
- For example, this query:

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

  can be represented as

  $$\sigma_{amount>1000}(\Pi_{user\_id,amount}(bird\_monthly\_bill)) \tag{1}$$

  and

  $$\Pi_{user\_id,amount}(\sigma_{amount>1000}(bird\_monthly\_bill)) \tag{2}$$

- The relational-algebra representation of a query only specifies partially how to evaluate a query
- To fully specify how to evaluate a query, the expression needs to be annotated with instructions specifying how to evaluate each operation
- The next step for the optimizer is to pick the most efficient query plan, which requires knowledge in the cost of each operation

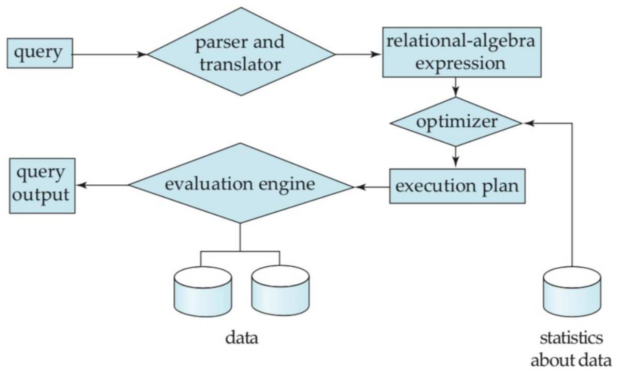


Figure 1: Typicall query workflow

- Things that the optimizer must account for is disk accesses, CPU time to execute a query, and in distributed systems, the cost of communication (Network I/O)
- Two important peices of data for this are the number of blocks transferred from disk and the number of random I/O accesses

Implementing the Select σ Operator

- `SELECT` is similar to `WHERE`, since to find the rows that match the `SELECT`, a linear scan must be done with a naive approach
- What if there `WHERE` clause contains the primary key, by definition only one record can match
- If everything is implemented with a B+ tree, then to find a record by equality, we have to search the tree, note that less than 1% of nodes are internal in a B+ tree

Sorting

- There are two important uses of sorting:
  1. The user can specify an `ORDER BY`
  2. Certain relational operators such as `JOIN` can be implemented more efficiently
- The common out-of-core sorting mechanism used in RDBMS is External Sort-Merge, or Two-Pass Multiway Mergesort, which allow for sorting relations larger than what can fit in memory
- To sort, first divide the relation into chunks of M blocks, then sort each chunk individually, then in RAM, set asside $M-1$ buffers for input, and 1 buffer for the eventual output to disk. Fill each of those buffers with B elements from their correspoding buffer, then merge the $M-1$ buffers and fill the last remaining buffer
  - Note that each entry that was copied to the output block buffer is then deleted from the input.
  - When the output buffer block fills up, it is written to the final result.
  - When an input buffer is empty, we read another B records from the corresponding input file.

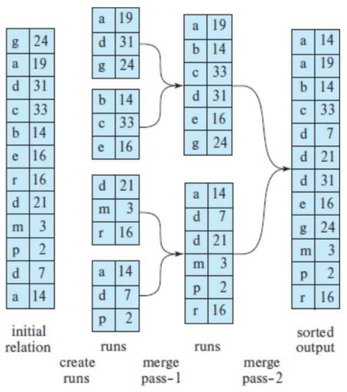


Figure 2: An example of this type of sorting

Join Operations $\bowtie_\theta$

- There are several main `JOIN` algorithms
  - Nested-Loop Join
  - Merge Joins
  - Hash Join
  - Map and Reduce-Side Joins
  - Shuffle Hash Join
  - Broadcast Hash Join
- **Nested Loop Join**
  - This is the simplest for computing $R \bowtie_\theta S$
  - This is essentially a nested for loop that compares every record R to every record in S, this runs in $\mathcal{O}(n^2)$ complexity with $n = \max(n_r, n_s)$
  - Some advantages are there is no need for an index, and there are no restrictions on θ, but this comes at the cost of speed

  - Ideally, one relation can fit in RAM, then we can leave that permanently cached for the nested loop while the othere relation can be loaded in from disk
  - If this isn't the case, each relation is broken up into blocks, then every block B_r is compared to every block B_s
  - If neither relation fits in main memory, it's best to use the smaller relation as the outer relation.
  - This can be sped up with the two following tricks:
    * If the join key itself is a primary key, we can stop the matching process as soon as a match is found.
    * When a nested loop is finished, we can then work backwards on the inner relation that way we reduce disk seeks
    * Can even create a temporary index on the inner relation if it isn't too costly to replace file-scans with index lookups
- **Merge Join R $\bowtie$ S**
  - Let $r(R)$ and $s(S)$ represent the relations we want to join, and $R \cap S$ their common attributes, then both relations are sorted along $R \cap S$
  - If the tables have been sorted by the join attribute, we only need to scan each table only once.
  - During the merge join operation, tuples are implicitly partitioned by key value.
  - If tuples do not fit in RAM for certain values of the join key, we can use a block nested join loop instead.
- **Hash Join**
  - We can partition the tuples of both relations R and S by hash value since we know that if h is a hash function and $t_r \in R$ and $t_s \in S$ and the join key K such that $K(t_r) = K(t_s)$, then $h(K(t_r)) = h(K(t_s))$
  - Let r_i and s_i be sets of tuples that all share the same join key hash, then for all i, tuples r_i are only compared to tuples in s_i
  - Hash joins can be used to improve the efficiency of the indexed nested-loop join.
  - If the relations are very large so that each relation is in a different node, then we can use a hash join with a Bloom Filter, so the hash of the key for each tuple in one relation is computed and added to the Bloom Filter
  - Since several hash functions are used for the Bloom Filter, there are a lot of collisions and thus the space required is small
  - This small Bloom Filter can then be easily sent over the network to the machine containing the other relation.
  - Then each key value in the other relation is checked, if the key does not exist in the Bloom Filter, we can throw out the tuple.
  - Only the tuples that the Bloom Filter thinks match on the join key are then sent to some machine to perform the actual join.
- **Joins in Spark**
  - Spark implements all these join algorithms on a distributed scale
  - The **shuffle hash join** (old default of Spark) shuffles tuples across the network so that all tuples with the same key value end up on the same node.
  - **Sort merge join** is more robust and faster than shuffle hash join and is the new default of Spark, this is similar to merge join in a RDBMS, except the algorithm is performed independently on each partition of data to prevent shuffling of data across the network. Data is transmitted over the network once if it must be relocated.

Other Set Operations

- For $L \cup R$, build a hash index I_i in RAM, where I_i is a specific hash partition, then add the tuples in r_i to the hash index only if not already in the index, then add the hash index tuples to the result
- For $L \cap R$, again build a hash index I_i, then for each tuple in r_i, probe the hash index and output the tuple if it is already present in the hash index.
- For $L - R$, again, build a hash index I_i, Then probe the hash index and if the tuple is present in an index, delete it. Once this is done return whats left

- Aggregations is implemented the same way as duplicate detextion, The tables are sorted or hashed except on join attributes, instead of removing duplicates, we apply some function to them

Query Optimization

- Golden Rule: Before doing an inner join: **filter**
- Given an expression in relational algebra, the query optimizer develops a query-execution plan that computes the same result as the relational algebra and also that is the least costly
- There are three steps to this process:
  1. Generating several equivalent relational algebra expressions
  2. Annotating the expressions with indices and statistics.
  3. Estimating the cost of each plan and then picking the one with the lowest cost

Step 1: Generating Equivalent Relational Algebra Expressions

- Two relational algebra expressions are equivalent if and only if on every legal database instance, both expressions generate the same set of tuples.
- Let E_i Be A Relation, L_i A List Of Attributes And θ_i Be A Set Of predicates
- Which one is faster?
$$\sigma_{\theta_1 \vee \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}) \tag{3}$$
- These equivalence rules and join guidance are used to transform some relational algebra expression E to equivalent expression E' in some set of equivalent expressions EQ.
- If E matches some equivalent rule R, we transform it to an equivalent expression $E'R(E)$, then update $EQ = EQ \cup E'$
- If we have E' for some E by applying an equivalence rule to some subquery e, then we know the rest of E and E' are identical
- The optimizer does not always need to generate all expressions. If the optimizer can estimate the cost associated with the plan that a particular generated expression will incur, we can prune it from the search.

Step 2: Estimating Statistics of Expressions

- The query optimizer does not know how large each table is without statistics about each table. The most basic statistic is the number of rows in the table.
- These statistics are based on samples, and are not updated on each mutation of the data.
- RDBMS usually catalog the number of tuples per relation, the number of blocks used to store a table, the size of a tuple for some relation (bytes), the number of rows that fit in one block, and the number of distinct values that appear in any relation by attribute
- In other databases, we can compute the statistics on tables with:

```sql
-- DB2
RUNSTATS ON TABLE <userid>.<table> AND INDEXES ALL

-- Oracle
RUNSTATS ON TABLE <userid>.<table> AND INDEXES ALL
```

- A lot of statistics and information is available in views named `pg_stat*` database in PostgreSQL.

Step 3: Choice of Evaluation Plan

- So we now have a large tree of possible evaluation plans based on equivalent relational algebra expressions as well as their costs based on statistics.

- An evaluation plan defines which algorithm should be used for each operation, and how the operations should coordinated
- The typical average query is a basic `SELECT ... FROM ... WHERE`, with 1-3 joins
- A reasonable assumption is that join order domiates cost. Suppose there are 3 relations, then the possible join orders are:

$$\begin{array}{llll}
r_1 \bowtie (r_2 \bowtie r_3) & r_1 \bowtie (r_3 \bowtie r_2) & (r_2 \bowtie r_3) \bowtie r_1 & (r_3 \bowtie r_2) \bowtie r_1 \\
r_2 \bowtie (r_1 \bowtie r_3) & r_2 \bowtie (r_3 \bowtie r_1) & (r_1 \bowtie r_3) \bowtie r_2 & (r_3 \bowtie r_1) \bowtie r_2 \\
r_3 \bowtie (r_1 \bowtie r_2) & r_3 \bowtie (r_2 \bowtie r_1) & (r_1 \bowtie r_2) \bowtie r_3 & (r_2 \bowtie r_1) \bowtie r_3
\end{array}$$

- With n relations, the number of different join orders is:

$$\frac{[2(n-1)]!}{(n-1)!} \tag{4}$$

- However, this can be reduced with dynamic programming by grouping together optimal join groups
- In bid data frameworks like Spark, it may be important to optimize your own queries
- There are also a few other heuristics for performing optimizations:
    1. Perform `SELECT` (`\sigma`) operations as early as possible, this reduces the size of intermediate results
    2. Perform projections (`\Pi`) early, since this reduces the amount of attributes passed around
    3. Cache queries that we use often

Lecture 12 - Functional Dependencies and Normalization 5/15/19

Functional Dependencies and Normalization

- Normalization is the process of refactoring tables to reduce redundancy in a relation. Normalization involves splitting a table with redundant data into two or more non-redundant tables.
- When there are redundancies in a table, we can decompose the table using functional dependencies.
- Problems with denormalized tables stem from redundancy, data integrity issues, and delay in creating new records
- Denormalized tables can be used for OLAP, data warehousing, and read-only read-heavy tables
  - We need to query aggregates FAST - denormalized tables are faster to query when we know exactly what we are searching for
  - Data integrity is not an issue, since its assumed another system did that
  - No delays in creating records because we do not create records in OLAP, the system does it for us on some schedule.

Functional Dependencies

- To determine if a table needs to be normalized, we define a series of rules called functional dependencies
- Given a relation R, with attributes $X \in R$, X is said to functionally determine another set of attributes $Y \in R$ if each value $X \in R$ is associated with one $Y \in R$. The R is said to satisfy the functional dependency $X \rightarrow Y$
- In the functional dependency $FD : X \rightarrow Y$ means that two tuples with the same value of X must also have the same value for Y, the converse is not true
- Rember that a **superkey** is a set of attributes K in a relation R such that no two tuples in the relation share the same values on all of the attributes K. There can be many superkeys in a relation
- If K is a superkey, and we take the projection of R over that superkey, it has the same cardinality as R

$$|\Pi_K R| = |R| \tag{1}$$

- A candidate key is a minimal superkey, with the smallest number of attributes, there can again be multiple candidate keys if they are independently unique, or there is a bijection mapping among the candidate keys
- Finally, the **primary key** is just the candidate key that is chosen to uniquely identify tuples in a relation
- Let α, β, γ be sets of attributes in a relation $r(R)$, then they have the following properties:
  1. **Reflexivity**: If $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$
  2. **Augmentation**: If $\alpha \rightarrow \beta$, then $\alpha\gamma \rightarrow \beta\gamma$
  3. **Transitivity**: If $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$
- From those first rules, the following can be derived:
  1. **Union**: If $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$ then $\alpha \rightarrow \beta\gamma$
  2. **Decomposition**: If $\alpha \rightarrow \beta\gamma$, then $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$
  3. **Psuedotransitivity**: If $\alpha \rightarrow \beta$, and $\Delta\beta \rightarrow \gamma$, then $\Delta\alpha \rightarrow \gamma$
  4. **Composition**: If $\alpha \rightarrow \beta$, and $\gamma \rightarrow \Delta$, then $\alpha\gamma \rightarrow \beta\Delta$

Atomic Attributes and First Normal Form

- Up till now in the course, attributes have been treated as atomic, indivisible units, so there is no unique key, they are flat and simple, and they are not collections
- If these conditions hold, we say this is in the first normal form (1NF)
- To convert a relation into 1NF, we add a primary key, remove any attributes that are collections and try to remove NULLs.

Second Normal Form (2NF)

- Decomposing a table down to 2NF allows for separation of concerns.
- A relation R is in 2NF if and only if for every attribute in R one of the following is true:
  1. The attribute appears in a candidate key;
  2. Is not partially dependent on the candidate key
- **Note**: R must also be in 1NF
- The following table shows which party won each state in a general election with State and Season as the candidate

A	B	C	D	E
Season	State	Winner	Birthdate	Capital
2016	California	Hillary Clinton	October 26, 1947	Sacramento
2016	Texas	Donald Trump	June 14, 1946	Austin
2012	Massachusetts	Barack Obama	August 4, 1961	Boston
2012	Michigan	Barack Obama	August 4, 1961	Lansing
2012	Arizona	Mitt Romney	March 20, 1947	Phoenix
2008	Oregon	Barack Obama	August 4, 1961	Salem
2008	Utah	John McCain	August 29, 1936	Salt Lake City

- This has the following functional dependencies
  1. $AB \to C$
  2. $C \to D$
  3. $B \to E$
  4. $E \to B$
  - **Note**: we also get $AB \to D$ via transitivity
- The candidate key here is the composite Season and State (AB).
- Birthdate has a partial dependency on the key. Birthdate depends on only Winner. This relation violates 2NF.
- To actually make this table compliant to 2NF, you would have to convert it like so:

A	B	C
Season	State	Winner
2016	California	Hillary Clinton
2016	Texas	Donald Trump
2012	Massachusetts	Barack Obama
2012	Michigan	Barack Obama
2012	Arizona	Mitt Romney
2008	Oregon	Barack Obama
2008	Utah	John McCain

C	D
Winner	Birthdate
Hillary Clinton	October 26, 1947
Donald Trump	June 14, 1946
Mitt Romney	March 20, 1947
John McCain	August 29, 1936

B	E
State	Capital
California	Sacramento
Texas	Austin
Massachusetts	Boston
Michigan	Lansing
Arizona	Phoenix
Oregon	Salem
Utah	Salt Lake City

Third Normal Form (3NF)

- For a table to be in **third normal form**, it must be 2NF, and not contain any attributes that are functionally dependent on any attributes that are not themselves a candidate key
- This also means that no attributes can be functionally determined by transitivity
- With the election table, we had that $AB \to C$ and $C \to D$, the dependency $C \to D$ is problematic since D is functionally determined by C which is not a candidate key. This violates 3NF, but works under 2NF
- **Closure**

- A closure F^+ of Fs is the full set of functional dependencies that can be logically implied given a set of functional dependencies and a set of axioms.
- Suppose we have that:
  1. $A \to B$
  2. $A \to C$
  3. $CG \to H$
  4. $CG \to I$
  5. $B \to H$
- We find the closure for A as follows:
  1. $A \to H$ using (1) and (5) transitivity
  2. $CG \to HI$ using (3), (4) and union rule
  3. $AG \to CG$ using (2) and augmentation
  4. $AG \to I$ using (4) and transitivity
- Then we have that:

$$F^+ = \{A \to H, CG \to HI, AG \to CG, AG \to I\} \tag{2}$$

- Relation schema R is said to be in third normal form with respect to F if, for every functional dependency in the closure F^+ of the form $\alpha \to \beta$, $\alpha \subseteq R$, $\beta \subseteq R$, at least one of the following is true:
  1. $\alpha \to \beta$ is a trivial functional dependency
  2. α is a superkey for R
  3. Each attribute A in $\beta - \alpha$ is explicitily containted in a candidate key for R

Making a Relation BCNF

- Let R be a relation that is not in BCNF. By definition, this means that D at least one non-trivial functional dependency $f : \alpha \to \beta$ such that α is not a superkey for R
- So we essentially split R into two relations: one with α and β, and one with α and everything except β
- When we decompose relations, we want to be sure that the decomposition is lossless.
- That is, if we decompose R into R_1 and R_2, we want

$$\Pi_{Attributes \in R_1}(R) \bowtie \Pi_{Attributes \in R_2}(R) = R \tag{3}$$

- And, given a set of functional dependencies, we can check that the decomposition is lossless:
  1. $R_1 \cup R_2 = R$
  2. $R_1 \cap R_2 \neq \{\}$
  3. $R_1 \cap R_2 \to R_1$ or $R_1 \cap R_2 \to R_2$, so $R_1 \cap R_2$ forms a superkey for either R_1 or R_2
- If we are going to decompose a large table into two or more smaller ones to reduce redundancy, we better make sure that we do not lose any information when we do the decomposition.
- A decomposition is said to be lossless if we can reconstruct the original relation R with a series of natural joins

$$R = R_1 \bowtie R_2 \bowtie ... \bowtie R_n \tag{4}$$

- When we decompose relations to BCNF or other forms, we want to study whether or not the decomposition is dependency preserving.
- Suppose R is decomposed into subrelations R_i, let F_i be the set of dependencies in F^+ that includes only attributes in R_i
  1. A decomposition is dependency preserving if

$$(F_1 \cup F_2 \cup ... \cup F_n)^+ = F^+ \tag{5}$$

  2. If the decomposition is not, then checking for updates that violate functional dependencies may require computing joins, which is expensive

Lecture 13 - Multivalued Dependencies and Transactions 5/20/19

Multivalued Dependencies (MVDs)

- Functional dependencies are only one way to decompose a relation into multiple normalized relations.
- MVDs express a condition among tuples of a relation R that represents many-to-many relationships within a key.
  - When this happens, certain attributes become independent of one another, so their values appear in all combinations
- MVDs are an assertion that two sets of attributes are independent of one another, but appear in the same relation
- MVDs are a generalization of functional dependencies
- The most common example of MVDs is with a Drinkers relation

  `Drinkers(name, address, emails, liquorsLiked)`
- Each drinker has one name and address, but may have several emails and liquors they liked
- The set of email addresses and liquors are independent. There is nothing special about one email address and one liquor
- If a drinker has 3 emails and 10 liquors they liked, then the drinker has 30 tuples in this relation, where each email is repeated 10 times
- **Note**: name $\to$ addr is the only redundancy with respect to functional dependencies
- A **multivalued dependency** $\alpha \twoheadrightarrow \beta$ (α multidetermins β) is a constraint between two sets of attributes and is an assertion that if two tuples of a relation agree on all attributes of α, then their components in β may be swapped amongst themselves and the result will be two tuples also in the relation
- This just says that we have all combinations of components in β in the tuple (in this case all email addresses paired with liquors, by person name), and that these components are independent of each other
- **Formal Definition of MVDs**: Let $r(R)$ be a relation, $\alpha \subseteq R$ $\beta \subseteq R$. The MVD $\alpha \twoheadrightarrow \beta$ holds on R if in any legal instance $r(R)$ for all pairs of tuples $t_1, t_2 \in r$ such that $t_1[\alpha] = t_2[\alpha]$, $\exists t_3, t_4 \in r$ such that
  1. $t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$
  2. $t_3[\alpha] = t_1[\alpha]$
  3. $t_3[R - \beta] = t_2[R - \beta]$
  4. $t_4[\alpha] = t_2[\alpha]$
  5. $t_4[R - \beta] = t_1[R - \beta]$
- In the drinkers example, we had that name$\twoheadrightarrow$email, which means for each drinker name, the set of email addresses appears in conjunction with the set of liquors liked

Properties of MVDs

1. An MVD $\alpha \twoheadrightarrow \beta$ is trivial if $\beta \subseteq \alpha$ or $\alpha \cup \beta = R$
2. **Complementation**: If $\alpha \twoheadrightarrow \beta$, then $\alpha \twoheadrightarrow R - \beta$, meaning β is a set independent of everything else in R
3. **Transitivity**: $\alpha \twoheadrightarrow \beta$, $\beta \twoheadrightarrow \gamma$, $\implies \alpha \twoheadrightarrow \gamma$
4. **Promotion**: Every FD is an MVD, so $\alpha \to \beta \implies \alpha \twoheadrightarrow \beta$
   - This is true since if the tuples agree on α, then swapping values of β between tuples does not change the semantic meaning of the tuples
5. **Augmentation**: also holds as does coalescence and replication as part of Armstrong's Axioms $\alpha \twoheadrightarrow \beta$

6. Note that **Splitting** ($\alpha \to \beta\gamma \implies \alpha \to \beta, \alpha \to \gamma$) does not hold, unlike with functional dependencies

Fourth Normal Form (4NF) and Higher Order Forms

- There is a lot of redundancy introduced by MVDs in the `Drinkers` example, we can use these MVDs to decompose the relation
- The functional dependency Name $\to$ Address also induces redundancy
- The fourth normal mode is essentially BCNF applied to MVDs
- A relation R is 4NF for all MVDs in F and F^+ if:
  1. $\alpha \twoheadrightarrow \beta$ is trivial, or
  2. α is a superkey for R
  3. R must also be BCNF given any functional dependencies
- If a relation is not 4NF given a set of FDs and MVDs, we can decompose R so that it is 4NF. First use all the FDs and normalize relations into BCNF, then once those have been applied, then for all the relations, check their MVDs, if an MVD $\alpha \twoheadrightarrow \beta$ is nontrivial and α is not a superkey, then decompose R into $R_1(\alpha, \beta)$ and $R_2(\alpha, \Omega)$, where Ω is all the attributes in R aside from α and β
- Note that very few relations in 4NF are also not 5NF, which means that every combination of β is in the table.
- Tables in 5NF cannot be decomposed further without becoming lossy.
- The Sixth Normal Form (6NF) just means that every join dependency in R is trivial
- Relations should be normalized to reduce redundancy and protect integrity during inserts updates and deletes
- It is possible to overnormalize and end up with sub-relations that require joins (expensive)
- 3NF is the only normal form that guarantees lossless decomposition that preserves dependencies
- Joins are expensive for read-only workloads. Start with normalized tables, and denormalize if we need better performance.

Transactions

- A transaction is a group of operations on data that form one logical unit.
- In SQL this is represented with the following:

```
BEGIN TRANSACTION
... # Data Transactions
END TRANSACTION
```

- Transactions are indivisible (atomic), either executed entirely or not at all
- During a transaction, the RDBMS will start executing data operations, if at any point of failure occurs:
  - Data Integrity Failure
  - Constraint Failure
  - Other Errors
  - Disk Failure
  - System Outage

  The transaction stops and undoes all the operations it just did with a rollback
- Even a single SQL query performs multiple data operations, but it is assumed to be atomic.
- This get complicated with concurrent users executing groups of SQL queries concurrently
- RDBMS must take special care to ensure transactions operate correctly
- A transaction must maintain database consistency, we do not want to lose data, add extraneous data, or have data with improper values

- A database must maintain **ACID** compliant transactions, which means that the transactions must
  1. **Atomicity**: The result of all operations in a transaction are represented in the database or none are
  2. **Consistency**: When transactions are executed in isolation, data remains consistent
  3. **Isolation**: When transactions executed concurrently, none of the transactions need to know about or depend on each other
  4. **Durability**: After the transaction completes successfully, the changes persist to the database, even if there are system failuresvalues
- A transaction that completes successfully is considered committed.
- After all steps have run successfully, but before the result has been written to the database, that transaction state is partially committed.
- A transaction is active while it is still processing data.
- If a transaction has failed, all changes are rolled back. The transaction is then in the aborted state.

Lecture 14 - Transactions and Locking 5/20/19

Transactions

- A transaction is a logical group of read and write operations
- This group must operate atomically, so it either all succeeds or fails
- When multiple transactions access and modify data concurrently, we can get a whole slew of problems with consistency, particularly when each transaction can write.
- What happens when a transaction T_1 tries to write to the same block at the same time as T_2?
- With n participating transactions, there are $n!$ possible serial schedules
- A schedule is conflict serializable if it is serial, or if it is equivalent in result to a serial schedule.
- If two instructions acting on different data points, we can swap the order of these two transactions without affecting the result
- If the two transactions are both reads, then order never matters. If the first is a read and the second is a write, then order matters; the same is true for vice versa.
- When two writes happen at the same time, both transactions will write to the value of X, so these transactions conflict
- Transactions will transform a database from a consistent state to another consistent state if there are no system failures, and the transactions are executed in isolation.

Schedule Serializability

- For each consecutive pair of operations from different transactions T_i and T_j in some schedule S, if the operations don't conflict, the order can be swapped to get a new schedule S'
- Operations continue to get swapped until there are no more conflicting operations
- A schedule is **serial** if T_i finishes all of its operations before T_j begins
- Terminology:
  1. S and S' are called **conflict equivalent** if we can convert S into S' by swapping non-conflicting operators
  2. Not all serial schedules are conflict equivalent.
  3. If a schedule S is conflict equivalent to a serial schedule, we call it conflict serializable.
  4. Two schedules can produce the same result but not be conflict equivalent.
- Consider some schedule S. A **precedence graph** G from S. The vertices consist of the individual transactions T_i, with 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
  3. T_i executes `write(X)` before T_j executes `write(X)`
- If $T_j \rightarrow T_j$ exists in the graph, then in any serial schedule S' equivalent to S, T_i must finish executing before T_j begins
- If there are no cycles in the graph, then S is conflict serializable

Transaction Isolation Levels and Implementation

- Isolation levels:
  1. **Serializability**: Serializable typically ensures serializable execution, easy to understand with limited concurrency
  2. **Repeatable Read**: only commited data may be read and no transaction may write between two reads of the same data point
  3. **Read Commited**: similar to Repeatable Read, but allows a transaction T_2 to update a data point in between reads of it in T_1 and then commit the change
  4. **Read Uncommited**: allows uncommited data to be read
- Regardless of isolation level, a transaction T_j should not modify a data point that has already been modified/written by another transaction T_i. - These are called **Dirty Writes**

- SQL has an option called autocommit, when disabled, the keyword `COMMIT` must be included to commit the result of the transaction, so each query needs a `COMMIT`
- Each transaction can aqcuire a lock for each data item it accesses
  - Locks are held long enough to ensure serializability, but not enough to degrade performance
- Each transaction has a timestamp associated with it. Each data item maintains two timestamps, one of the most recent read and one of the most recent write
- The transaction then accesses each data item in order of the timestamps if two transactions conflict
- The transaction only reads the local copy of the database which is private to the transaction, and is thus isolated from other transactions.
- Then, all modifications within the transaction are made on the transaction's private copy of the data
- The transaction engine stores a record of what changes need to be made to the database. After the transaction succeeds, the engine then checks if any other transaction modified the same data items. If it did, the transaction aborts and we rollback. Otherwise, the transaction commits.
- The major advantage of this is that you never have to wait to write data, and reads never are aborted, the disadvantage however is none of the transactions can see the other's writes, we will not know about this until we try to commit
- Snapshot isolation is very expensive and only worth if the probability of write conflicts are low

Locking

- The most common way to ensure isolation is with locks
- While one transaction is accessing data, no other transaction can modify that data
- **Shared:**: If a transaction holds a shared-mode lock on some data item P, it can read from the item, but not to write it.
- **Exclusive**: If a transaction holds an exclusive-mode lock on some data item P, it can both read from and write to the item.
- Each time a transaction needs access to a data item, it must request a lock
- Multiple transactions can hold a read lock on the same data item, but as soon as a transaction wants to write to that item, it must wait until all shared-mode locks are released
- Starvation is prevented by whenever a transaction T_i requests a lock on data item A in a particular mode M, the lock manager will grant the lock if:
  1. No other transaction holds a lock on A that conflicts with M
  2. There is no transaction that is waiting for lock A and made its request before T_i
- When a lock has complete using item P, it will release its lock, but potentially not immediately, releasing too early loses serializability, since other transacitons can make clobbering writes, if we unlock too late, transactions will be unreasonably blocked, and if we unlock too late, the transactions get unreasonably blocked
- Another problem stems from deadlock, since if we unlock the lock too soon we get inconsistent states, but unlocking too late will get you deadlock
- The lock manager can also determine when a deadlock is about to occur, and tell the transaction to rollback.
- An important locking strategy that ensures serializability is the **Two-Phase Locking Protocol (2PL)**: which requires that each transaction makes lock and unlock requests in two phases:
  1. **Growing Phase**: The transaction may require locks, but not release any
  2. **Shrinking Phase**: The transaction may release locks, but not aquire any
- This strategy may result in transactions for a long time, increasing isolation and serializability, but decreasing performance
- Once deadlock is detected, the following procedure is performed:
  1. Choose a transaction that will be rolled rollback, this can depends on how long a transaction has been running, how much data it has and needs, and how many other transacitons will be rolled back in cascade
  2. Rollback the transaction
  3. Repeat until no deadlock

- Instead of requesting locks on individual data items, we can request locks on a group of data items based on some tree-like hierarchy.
- Similarly, we lock the entire table or database by requesting a lock on the root, and this implicitly locks all data in the table or database.

Lecture 15 - Stream Processing 2019-05-29

Stream Processing

- All the systems for data seen are materialized, so they either store all the data or they have access to all the data
- Map-Reduce and ETL jobs read in a set of input files and produce a new set of output files. This data may then be written to RDBMS or Data Warehouses.
- The data is of a known and finite size. It's bounded. We know what the first record is, and what the last record is. It is a countable set.
- However, this is not always the case, a lot of the data is unbounded, that is, it arrives as it is produced, the dataset is never complete
- We can process data in windows of time such as a day or hour, etc
- At the beginning of the window, we clear a buffer and start collecting data, when the window expires, the daa is processed and perhaps written to the RDBMS
- The problem with this approach is what happens if we need to do analysis across windows? The data isn't available until the window closes
- **Data Streams**
  - In RDBMS and batch processing, the input is usually a bunch of files that are then divided into records before the processing occurs
  - In stream processing, it's usually individual records, called events and each event is partially identified with a timestamp.
- In the streaming system, there are generally two ends:
  1. **Publisher**: the node or process that generates the event
  2. **Subscriber**: consumer (or recipient): a node or process that receives events and does something with them.
- The system works by the publisher continuously generating events into the RDBMS while the subscriber constantly polls the RDBMS for new events
- Polling is expensive, so we want the system to "push" notifications to nodes, this can be done by triggers, but this is limited
- The other option is push notifications, where in between the publisher(s) and subscriber(s) lies a message bus, some connection between them.
- Unlike Unix pipes, a message bus in a streaming system has multiple publishers and multiple subscribers.
- Unlike batch processing that has a strict data reliability gaurantee, streaming systems are best effort, so if producers send messages faster than consumers can process them, it then becomes implementation specific
- When we want super low latency and are OK with losing some messages in the process, we can use UDP and encode reliability at the application layer.
- If we are can tolerate higher latency and require perfect reliability, use TCP.
- Instead, we can have a special database server called a "message broker" that sits in between the publishers and subscribers.
  - Here, publishers write messages to the broker, while subscribers read messages to the broker
- The broker can handle the case where nodes come and go. Durability is now handled by the broker rather than each node.
- Differences between a message broker and RDBMS
  1. RDBMS keeps data until it is deleted by the user. Message brokers delete data once it is delivered to the ubscribers. Not good for long term storage
  2. Low throughput. If there is a bottleneck in subscribers, performance degrades.
  3. RDBMS supports indexes for finding data. While brokers support some way of extracting particular topics, it is not as sophisticated as index
  4. RDBMS queries are based on snapshots of data. Brokers do not accept arbitrary queries, but they DO notify subscribers when data changes.
- If there are multiple subscribers, how does the broker determine which one gets the message?

1. **Load Balancing**: the broker chooses one subscriber to receive the message either arbitrarily or basd on some shard or partition key.
2. **Fan Out**: the broker delivers the message to ALL subscribers in particular groups. Then, each node may do something different with the message. (lambda architecture)

- Once the subscriber is done processing a message, it sends an ACK back to the broker to acknowledge receipt and processing of the message.
- Since the ACK may be lost in the network, the message broker uses a two-phase commit protocol (2PC) to manage the situation
- The transaction only commits once the ACK has been recieved by the broker

Load Balancing + Retransmits = Out of Order

- Usually, a message is processed in the same order as it was delivered by the publishers, but we can have a situation where messaes are processed out of order when using redelivery with load balancing
- Once the broker receives an ACK from the subscribers, the message is deleted from the broker.
- Brokers can instead use recovery logs to maintain order. In this case, a publisher sends a message to a broker and the broker appends the message to the end of a log.
- The subscriber then reads the log sequentially (`tail -f`). Within each topic, the logs can be partitioned to increase parallelism
- Within each partition, the broker maintains an integer offset for each message in the log. The subscriber and the broker also maintains an offset denoting the last message that was processed.
- Disk I/O is slow, but by partitioning these logs over many disks, we can allow fast throughput.
- Typically, the broker will assign an entire partition to a single subscriber and then that subscriber reads in the messages from the log in a single-threaded manner.
- Under the log-based model, the broker does not need to keep track of ACKs. Both the subscribers and the broker are synchronized via the offsets.
- To prevent exhausting disk space, the logs are divided into segments and after a period of time, segments are deleted from disk.
- If a subscriber falls so far behind the publishers that the next offset is part of the log that was deleted, we lose messages.
- The main purpose of log-based message brokers is to solve problems that occur when subscribers disappear and there is still work to be done, and to alleviate issues involving messages being processed out of order.
- In a mature software company, there are multiple copies of the data in many different types of databases, and these all need to be synchronized; this can be done with a streaming system
- To search a stream, we instead specify the search query a priori. We will miss all past events that match the query, but we will be able to percolate up any events that happen after the query is issued.

Lecture 16 - NOSQL 2019-06-04

NOSQL

- The most common models used for NoSQL are
  1. Document store (MongoDB, CouchDB)
  2. Graph (Neo4j, OrientDB, Giraph)
  3. Key-value store (Redis, Memcached)
  4. Columnar (Cassandra, HBase, DynamoDB)
- CAP Theorem (Brewer's theorem): impossible for a distributed data store to simulatenously provide more than two of the following guarantees:
  1. **Consistency**: Every read receives the most recent write, or an error is thrown.
  2. **Availability**: Every request receives some kind of response (not an error), without guaranteeing the most recent write. It always returns a response.
  3. Partition tolerance: The system continues to function even if messages are lost or are delayed by the network between or among nodes
- Sharding just means that we treat groups of rows as one unit and distribute them across the network to different data nodes potentially running different instances of the RDBMS.
- Once a shard is determined, it is independent of every other shard in the system.
- The problem with sharding is the increased latency and consistency and durability issues due to the distributed nature of the system, as well as the more complicated SQL required to handle this type of querying

MongoDB

- It is a document store which means that it ingests full documents into a JSON-like schema and allows querying on this schema.
- All NoSQL databases have tradeoffs different compared to the relational model.
- **Indexing**: Supports both primary and secondary indices, but not a B+tree only B-trees. Data are nested natively via JSON.
- **Ad Hoc Querying**: on field, range and regular expression, and can also include JavaScript.
- **Aggregation**: is supported like in SQL, but uses MapReduce to distribute embarrassingly parallel computations.
- **ACID Transactions**: are supported as of v4.0 and uses snapshot isolation rather than locks.
- **Data Format**: Data is basically just a big JSON blob with no particular format or schema on disk.
- **File Storage**: MongoDB can be "turned inside out" and used as a file system called GridFS.
- MongoDB ensures high availability by using replica sets, which are similar to data snapshots
- If the primary replica fails, an election process is conducted to determine which secondary should be the primary. Secondaries can also be used for high available reads.
- In MongoDB, if multiple documents are included in a write operation each document write is atomic, even if that document contains embedded documents, but the multi-document update itself is not atomic.
- An ODM (or ORM for RDBMS) is an API that allows users to use regular function calls to perform database inserts, updates, deletes and retrievals.
- OxMs generally support CRUD operations:
  1. **C**reate
  2. **R**ead

3. **U**pdated
4. **D**elete

- Using SQL or whatever DML the database supports directly will yield the best performance.
- Using an ODM or ORM does induce a small performance hit, but allows tighter integration into applications and is usually easier to maintain.
- Connecting Mongoose to MongoDB

```
//Import the mongoose module
var mongoose = require('mongoose');

//Set up default mongoose connection
var mongoDB = 'mongodb://127.0.0.1/my_database';
mongoose.connect(mongoDB, { useNewUrlParser: true });

//Get the default connection
var db = mongoose.connection;

//Bind connection to error event (to get notification of
// connection errors)
db.on('error', console.error.bind(console,
'MongoDB connection error:'));
```

- We first tell Mongoose about the MongoDB schema of a document collection using the Schema object and then compile it into a model.

```
// Define schema
var SomeModelSchema = new Schema(
{
    name: String,
    binary: Buffer,
    living: Boolean,
    updated: { type: Date, default: Date.now() },
    age: { type: Number, min: 18, max: 65, required: true },
    mixed: Schema.Types.Mixed, // arbitrary type
    _someId: Schema.Types.ObjectId, // unique object ID
    array: [], // empty array of unknown type
    ofString: [String], // array of type String
    // You can also have an array of each of the other types too.
    nested: { stuff: { type: String, lowercase: true, trim: true } }
})

// Compile model from schema
var SomeModel = mongoose.model('SomeModel', SomeModelSchema );
```

- MongoDB also allows for the addition of **Validators**

```
var breakfastSchema = new Schema({
    eggs: {
        type: Number,
        min: [6, 'Too few eggs'],
        max: 12,
        required: [true, 'Why no eggs?']
    },
    drink: {
        type: String,
        enum: ['Coffee', 'Tea', 'Water',]
    }
});
```

- Mongoose also supports **virtuals**. These are synthetic fields that are not persisted in MongoDB but are used for string formatting and other front end uses. They are similar to `struct`
- The model is a constructor that takes a schema definition constructs documents.
- CRUD operations are asynchronous, so you can write inserts like this:

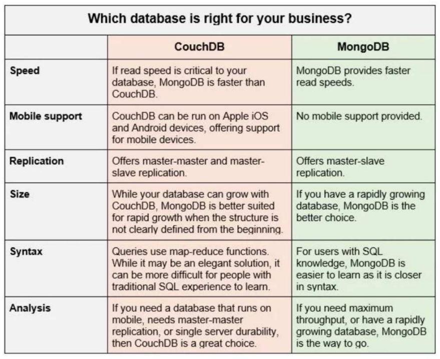
  ```
  SomeModel.create({ name: 'also_awesome', age: 20 },
      function (err, awesome_instance) {
          if (err) return handleError(err);
          // saved!
  });
  ```

- To retrieve records with Mongoose:

  ```
  var Athlete = mongoose.model('Athlete', yourSchema);

  // find all athletes who play tennis, selecting the 'name'
  // and 'age' fields
  Athlete.find({ 'sport': 'Tennis' }, 'name age',
      function (err, athletes) {
          if (err) return handleError(err);
          // 'athletes' contains the list of athletes that match the criteria.
  })
  ```

- MongoDB vs. CouchDB
  - CouchDB is another similar object store.
  - MongoDB uses a variant of JSON called BSON whereas CouchDB uses JSON.
  - MongoDB is truly schema-free whereas CouchDB conforms more to the document store model with structure
  - MongoDB allows strong consistency and expressive query language
  - MongoDB supports indexes natively (primary and secondary, B-tree), whereas CouchDB uses views as an index insteadD
  - CouchDB treats data as documents consisting of fields and attachments
  - CouchDB uses optimistic updates and lockless
  - CouchDB uses a REST API whereas MongoDB uses uses an ODB called Mongoose.
  - Both use MapReduce. MongoDB uses it for querying, CouchDB generates a key and value right after the document is created and is immutable unless the document changes.
  - Both use similar clustering methods,



Which database is right for your business?

	CouchDB	MongoDB
Speed	If read speed is critical to your database, MongoDB is faster than CouchDB.	MongoDB provides faster read speeds.
Mobile support	CouchDB can be run on Apple iOS and Android devices, offering support for mobile devices.	No mobile support provided.
Replication	Offers master-master and master-slave replication.	Offers master-slave replication.
Size	While your database can grow with CouchDB, MongoDB is better suited for rapid growth when the structure is not clearly defined from the beginning.	If you have a rapidly growing database, MongoDB is the better choice.
Syntax	Queries use map-reduce functions. While it may be an elegant solution, it can be more difficult for people with traditional SQL experience to learn.	For users with SQL knowledge, MongoDB is easier to learn as it is closer in syntax.
Analysis	If you need a database that runs on mobile, needs master-master replication, or single server durability, then CouchDB is a great choice.	If you need maximum throughput, or have a rapidly growing database, MongoDB is the way to go.

Figure 1: CouchDB vs MongoDB