EECS-317 Data Management and Information Processing

Lecture 18 – Course Review

Steve Tarzia

Spring 2019

Northwestern

Announcements

- Final exam on Thursday!
- Practice Final exams posted.
- All homework answers are posted.
- Project due Wednesday, June 12th.

These are a few of my favorite slides

Why use a relational database?

- **Scalability** – work with data larger than computer's RAM
- **Persistence** – keep data around after your program finishes
- **Indexing** – efficiently sort & search along various dimensions
- **Integrity** – restrict data type, disallow duplicate entries
- **Deduplication** – save space, keep common data consistent
- **Concurrency** – multiple users or applications can read/write
- **Security** – different users can have access to specific data
- **"Researchability"** – SQL allows you to concisely express analysis

Sometimes we start with one redundant table and break it down to reflect the logical components

staff					
id	***name***	***department***	***building***	***room***	***faxNumber***
11	Bob	Industrial Eng.	Tech	100	1-1000
20	Betsy	Computer Sci.	Ford	100	1-5003
21	Fran	Industrial Eng.	Tech	101	1-1000
22	Frank	Chemistry	Tech	102	1-1000
35	Sarah	Physics	Mudd	200	1-2005
40	Sam	Materials Sci.	Cook	10	1-3004
54	Pat	Computer Sci.	Ford	102	1-5003

This is called *Normalization*

staff		
id	***name***	***department***
11	Bob	1
20	Betsy	2
21	Fran	1
22	Frank	4
35	Sarah	5
40	Sam	7
54	Pat	2

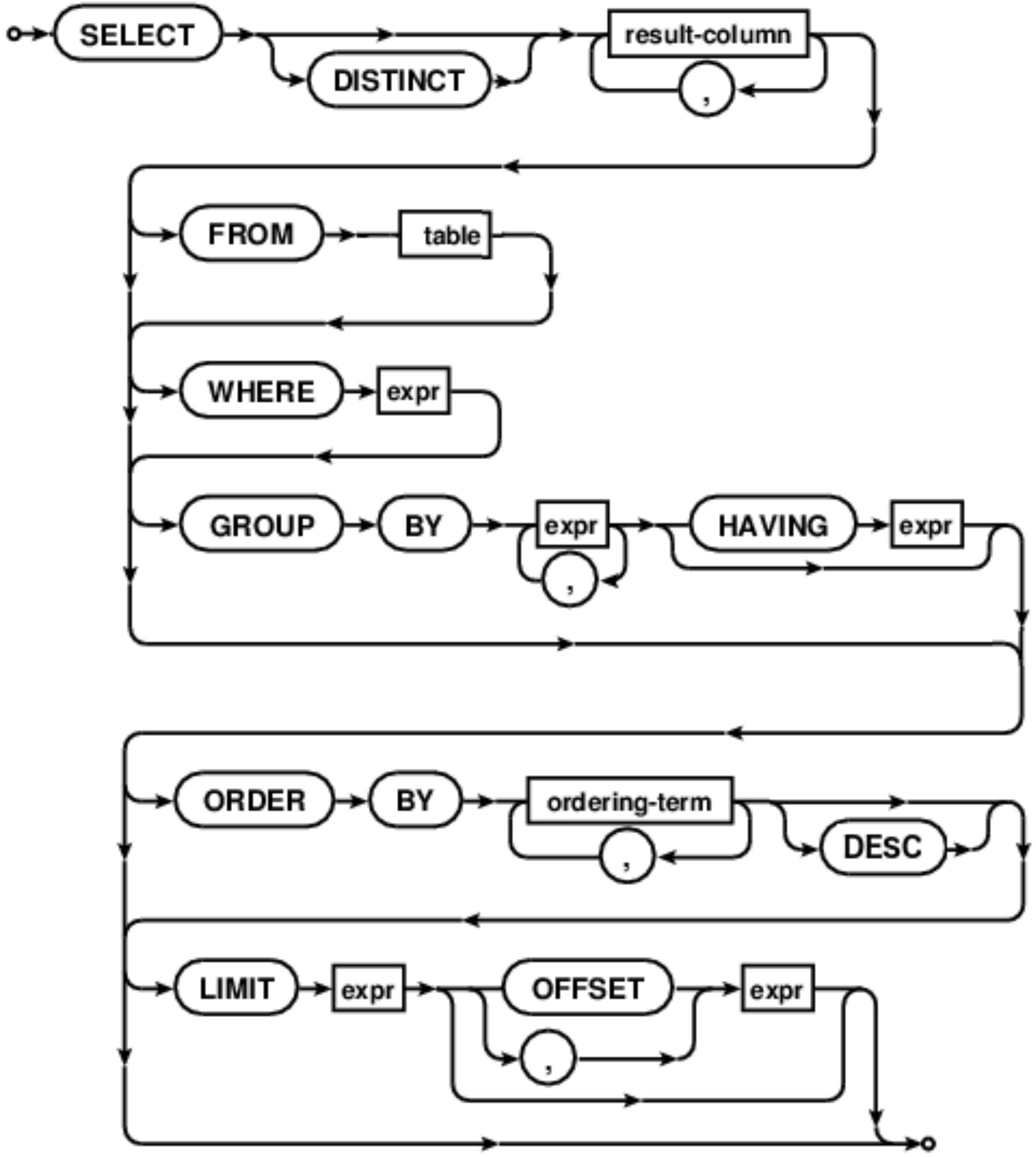
department		
id	***name***	***building***
1	Industrial Eng.	1
2	Computer Sci.	2
4	Chemistry	1
5	Physics	4
7	Materials Sci.	5

building		
id	***name***	***faxNumber***
1	Tech	1-1000
2	Ford	1-5003
4	Mudd	1-2005
5	Cook	1-3004
6	Garage	1-6001

- Removes redundancy
  - Save space
  - Edit values in one place, so duplicates don't become inconsistent
- Tables can be populated separately
- **But**, you are adding a new *id* column for each table

Syntax diagrams

- Any path from start to end is a valid statement.
- Choose which arrows to follow
- The rectangles refer to other diagrams.
- Used by our SQL book
- Used by SQLite online docs: https://sqlite.org/lang.html



SELECT steps (abbreviated)

1. `FROM` chooses the table of interest
2. `WHERE` throws out irrelevant rows
3. `GROUP BY` identifies rows to combine
4. `SELECT` tells what values to return (allowing math and aggregation)
5. `HAVING` throws out irrelevant rows (after aggregation)
6. `ORDER BY` sorts
7. `LIMIT` throws out rows based on their position in the results

Each step gets closer to the specific result you want.

GROUP BY explained

- GROUP BY combines multiple rows into one row in the result.
- Rows with the same value for the *grouping criterion* are grouped.
- An aggregation function is usually applied.

```
SELECT CategoryID, COUNT(*) AS category_count,
       MAX(RetailPrice) AS most_expensive_price
  FROM Products GROUP BY CategoryID;
```

ProductName	RetailPrice	CategoryID
Trek 9000 Mountain Bike	1200	2
Eagle FS-3 Mountain Bike	1800	2
Dog Ear Cyclecomputer	75	1
Victoria Pro All Weather Tires	54.95	4
Dog Ear Helmet Mount Mirrors	7.45	1
Viscount Mountain Bike	635	2
Viscount C-500 Wireless Bike Computer	49	1

CategoryID	category_count	most_expensive
1	18	179
2	4	1800
3	4	85
4	9	189
5	2	180
6	3	34

How JOIN builds a composite table

```
SELECT * FROM staff JOIN department
              ON staff.departmentId=department.id
```

Start with the first table (staff)

Join with rows from the 2nd table (department) that match according to the `ON` columns

staff.*id*	staff.*name*	*staff.room*	staff.*departmentId*	department.*id*	department.*name*	department.*buildingId*
11	Bob	100	1	1	Industrial Eng.	1
20	Betsy	100	2	2	Computer Sci.	2
21	Fran	101	1	1	Industrial Eng.	1
22	Frank	102	4	4	Chemistry	1
35	Sarah	200	5	5	Physics	4
40	Sam	10	7	7	Materials Sci.	5
54	Pat	102	2	2	Computer Sci.	2

Using INNER JOIN, what if rows don't match one-to-one?

staff			
id	***name***	***room***	***departmentId***
11	Bob	100	1
20	Betsy	100	2
21	Fran	101	1

department		
id	***name***	***buildingId***
1	Industrial Eng.	1
2	Computer Sci.	2
4	Chemistry	1
1	Physics	4
1	Materials Sci.	5

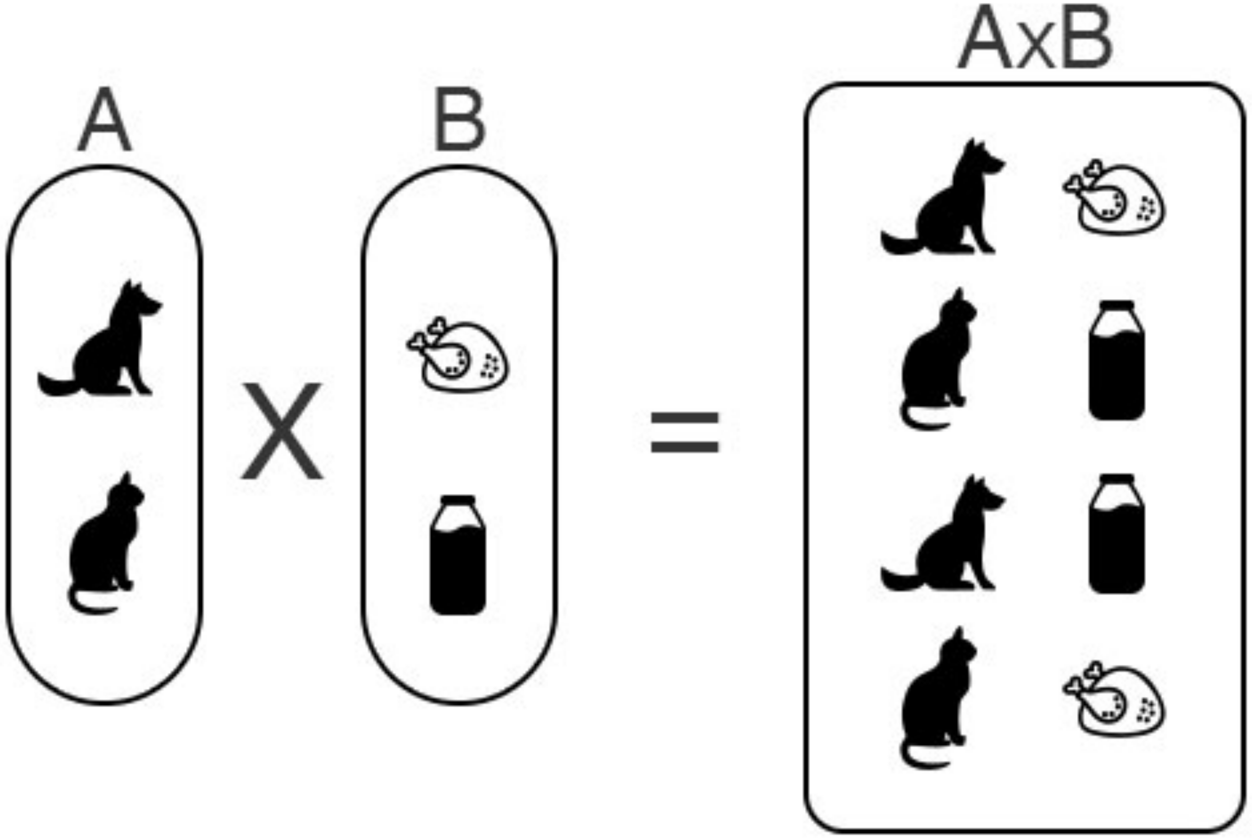
```
SELECT * FROM staff JOIN department
ON staff.departmentId=department.id
```

In output,

- multiple matches leads to multiple rows.
- no matches leads to no rows

staff.***id***	staff.***name***	***staff.room***	staff.***departmentId***	department.***id***	department.***name***	department.***buildingId***
11	Bob	100	1	1	Industrial Eng.	1
11	Bob	100	1	1	Physics	4
11	Bob	100	1	1	Materials Sci.	5
20	Betsy	100	2	2	Computer Sci.	2
21	Fran	101	1	1	Industrial Eng.	1
21	Fran	101	1	1	Physics	4
21	Fran	101	1	1	Materials Sci.	5

CROSS JOIN is like the **cartesian product** of two sets



Cartesian Product of Two Sets.

- Take every element (row) of the first set (table) and combine it with every element of the second set.
- If first set has N elements and second set has M elements, then cartesian product has N·M elements.
- There is no "ON" expression to limit results:
  - `SELECT Orders CROSS JOIN Order_Details;`

NATURAL JOIN

- A shorthand notation to make some JOINs shorter to express.
- NATURAL JOIN matches rows using whatever columns have identical names.

For example:

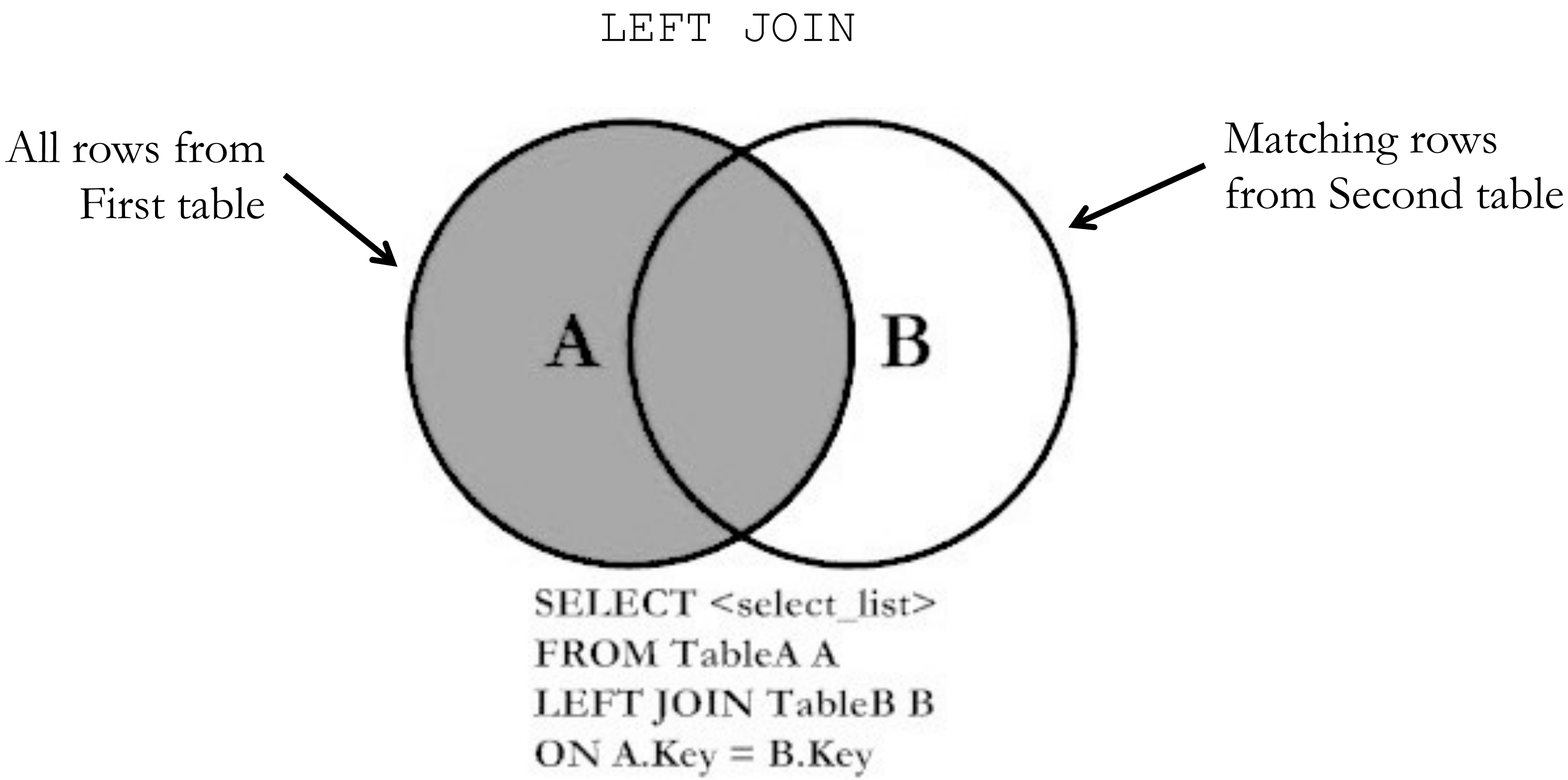
```
SELECT * FROM Orders JOIN Order_Details
  ON Orders.OrderNumber=Order_Details.OrderNumber;
```

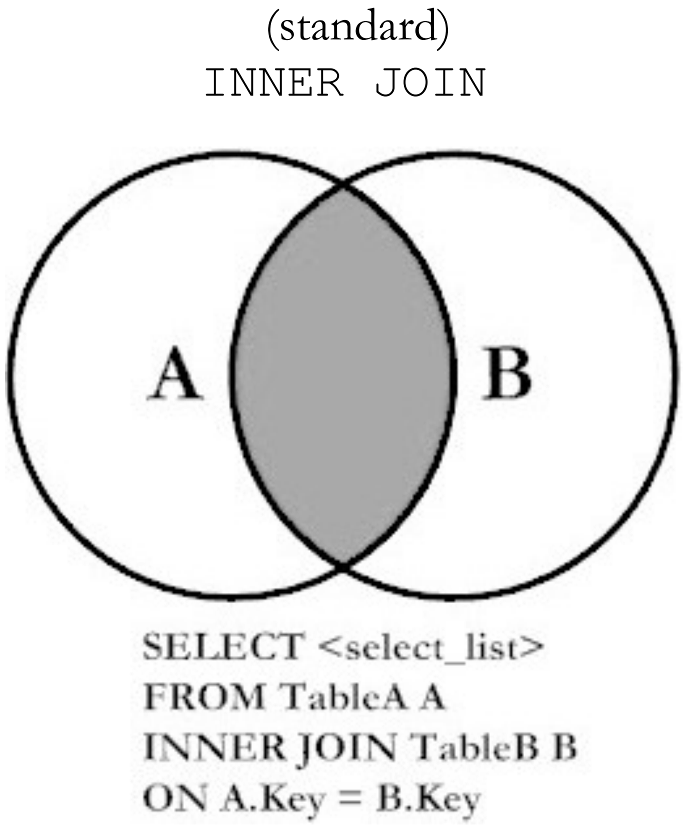
Becomes:

```
SELECT * FROM Orders NATURAL JOIN Order_Details;
```

LEFT JOIN

- LEFT JOIN includes **all** rows in the first table (***left***-hand side) and just the matching rows in the second table (right-hand side).





staff			
id	***name***	***room***	***departmentId***
11	Bob	100	1
20	Betsy	100	*NULL*
21	Fran	101	1
22	Frank	102	99999
35	Sarah	200	5
40	Sam	10	7
54	Pat	102	2

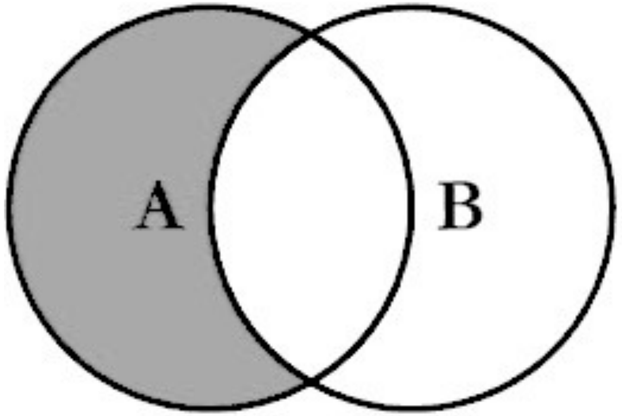
department		
id	***name***	***buildingId***
1	Industrial Eng.	1
2	Computer Sci.	2
5	Physics	4
7	Materials Sci.	5

- Betsy and Frank have NULLs in the right haft of the output because no matching department was found.
- In other words no pair of rows was found to satisfy the ON staff.departmentId=department.id

```
SELECT * FROM staff LEFT JOIN department ON staff.departmentId=department.id;
```

staff.***id***	staff.***name***	***staff.room***	staff.***departmentId***	department.***id***	department.***name***	department.***buildingId***
11	Bob	100	1	1	Industrial Eng.	1
20	Betsy	100	*NULL*	***NULL***	***NULL***	***NULL***
21	Fran	101	1	1	Industrial Eng.	1
22	Frank	102	99999	***NULL***	***NULL***	***NULL***
35	Sarah	200	5	5	Physics	4
40	Sam	10	7	7	Materials Sci.	5
54	Pat	102	2	2	Computer Sci.	2

LEFT JOIN with exclusion




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

- Includes rows from a table that ***must not*** match another table.
- Useful for finding rows lacking something.
- Just add a `WHERE` clause to look for *NULL* values in the right-hand side of the joined table
- For example, to determine which faculty members should be assigned a class:
  - `SELECT * FROM Faculty NATURAL LEFT JOIN Faculty_Classes` **`WHERE ClassID IS NULL`**`;`
- Which classrooms are unused?
  - `SELECT * FROM Class_Rooms NATURAL LEFT JOIN Classes` **`WHERE ClassID IS NULL`**`;`

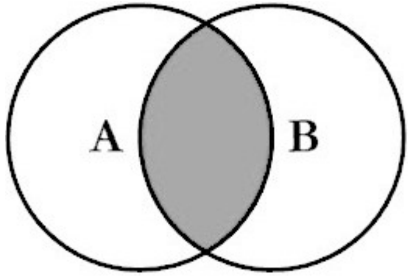
JOIN TYPES

Introduced different types of JOINs:

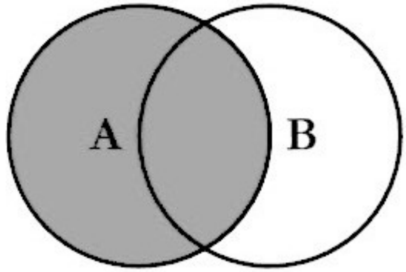
- **INNER** (default): prints all pairs of rows (one from first table, one from second table) that satisfy the *JOIN predicate*.

- **LEFT**: same as INNER, but adds rows from LEFT table that never satisfied the JOIN predicate.

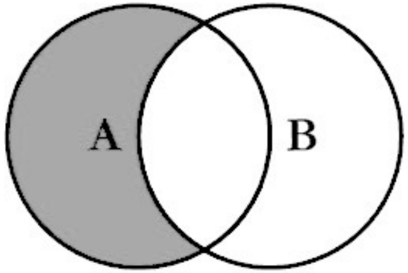
- **LEFT with exclusion**: only print rows form left table that never satisfied the JOIN predicate.




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

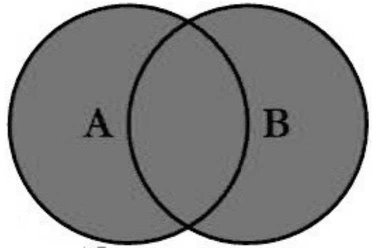



```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```

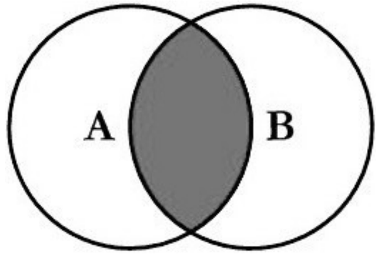



```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```

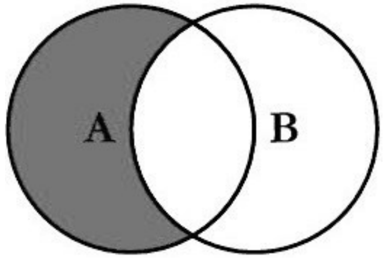
UNION, INTERSECT, and EXCEPT
are used to combine two SELECT statements



- **`UNION`** prints rows from *either of two* `SELECT`s (printing duplicates just once)



- **`INTERSECT`** prints rows *present in both* `SELECT`s

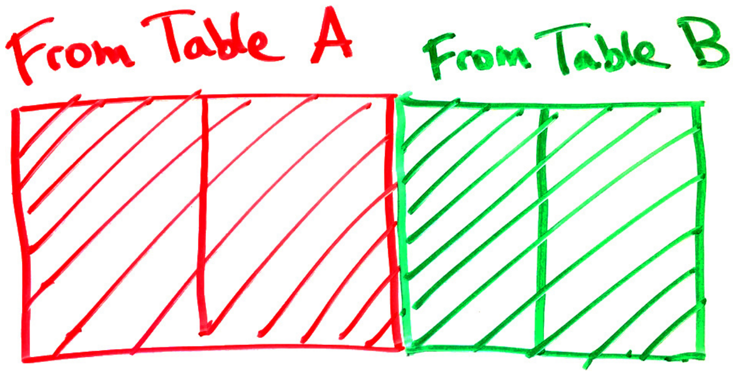


- **`EXCEPT`** prints rows *present in one* `SELECT` but *missing from another* `SELECT`

JOIN vs. **UNION**

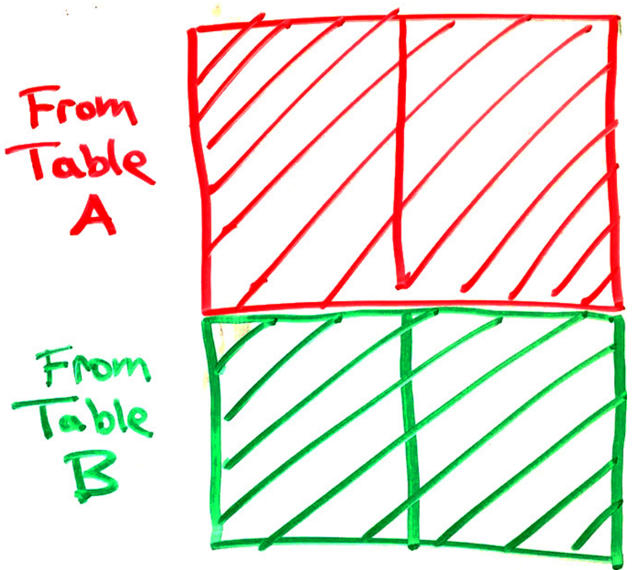
- JOINs combine tables *horizontally*.
  - Match rows from two tables based on one or more columns matching.
  - Creates a wider set of rows, adding **columns** from both tables.

JOIN:



- UNION, INTERSECT, and EXCEPT combine result tables
  - Number & type of columns in the two result tables must match
  - Changes the number of **rows**, not columns

UNION:



Summing an indicator variable

Two ways to count recipes with "salsa" in description:

- `SELECT COUNT(*) FROM Recipes WHERE` **`RecipeTitle LIKE "%salsa%";`**
  - `WHERE` clause keeps just the rows matching "salsa," then these rows are counted.
- `SELECT SUM(`**`RecipeTitle LIKE "%salsa%"`**`) FROM Recipes;`
  - A column is created for every recipe indicating whether its title matches "salsa" or not.
  - Column's value will be **1** if it matches and **0** if not.
  - Sum of all the ones and zeros will be the count of matching recipes.
- First approach is easier to understand, but second is shorter.

CASE conditional

WHEN condition is tested for every row, giving *true* or *false*

```
SELECT CASE WHEN CategoryID=2
THEN  "Bike"
ELSE  ProductName  END FROM Products;
```

If condition is *true* then use the first value.

If condition is *false* then use the second value.

Output:

1	Bike
2	Bike
3	Dog Ear Cyclecomputer
4	Victoria Pro All Weather Tires
5	Dog Ear Helmet Mount Mirrors
6	Bike
7	Viscount C-500 Wireless Bike Computer
8	Kryptonite Advanced 2000 U-Lock
9	Nikoma Lok-Tight U-Lock

Regular Expressions

- Regular expressions are used to match text, both in SQL and in many other data management tools.
- A match anywhere in the text returns *true*.
- **^** anchors to the beginning
- **$** anchors to the end
- **.** matches any character
- **[…]** specifies a set of possible characters
- **[a-z]** hyphen specifies a range
- **[^abc]** carrot within brackets negates the match

- Repetitions are supported:
  - ***** any number
  - **+** one or more
  - **?** zero or one
  - **{n,m}** n to m repetitions
- **|** pipe character gives OR
- **(…)** can be used for grouping

Examples

`^1.*t$` *matches:* "12 point", "100.3 feet", "111ttt", "1t"
does not match: "This 12 point font"

`[Cc]ats?` *matches:* "Cat behavior", "5 cats", "catnip"
does not match: "cast", "CATS"

`[0-9]+.[a-z]?` *matches:* "249032/b", "23.", "
does not match: "a", "aa", "1"

`([cC]at|[Dd]og)( food)?` *matches:* "cat food", "Dog"
does not match: " food"

Why sorting is not enough

- You can't sort in **multiple dimensions**
  - Let's say you want to find a product quickly according to either it's name, manufacturer, or price. You can only sort by one of the there three columns.
- Can't **insert new data** without shifting everything over to make room.
- It doesn't take advantage of the hardware's storage hierarchy.
  - The binary search will have to access the disk in every step because the index is distributed over the full data set.
  - It would be better to put all the index data close together (spatial locality).

The solution:

- **Indexes** provide quick to rows by looking up column values.

When to index columns?

- When a query is slow!
- Generally, add an index if the column is:
  - Used in `WHERE` conditions, or
  - Used in `JOIN … ON` conditions, or
  - A foreign key refers to it.
- Also helpful if the column is:
  - In a `MIN` or `MAX` aggregation function

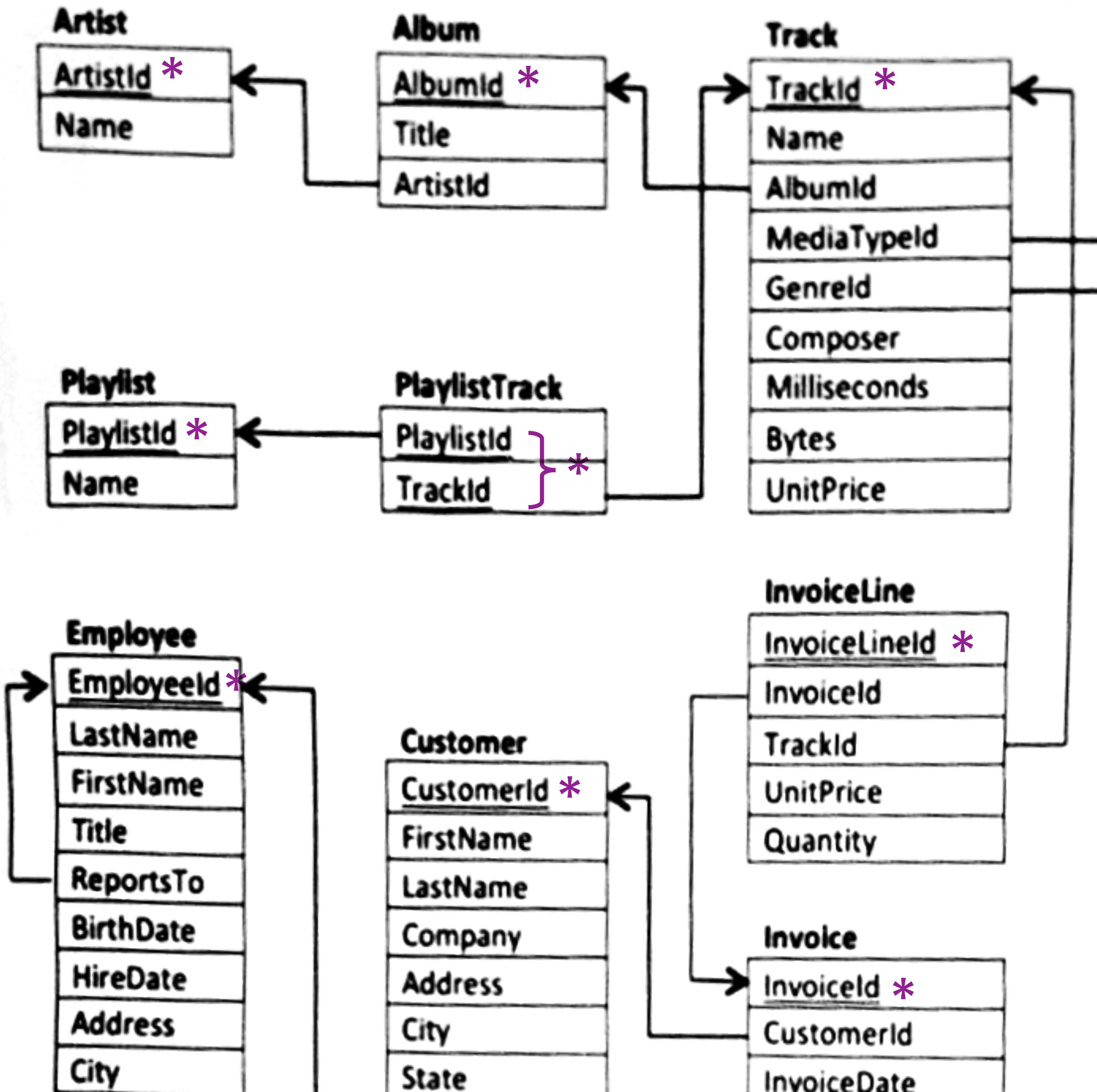
Key and Index terminology in SQL

- Plain **key** or **index** is just a way to find rows quickly
  - Just creates a search tree.
- **Unique key** is an index that prevents duplicates
  - Bottom level of search tree has no repeated values
  - DBMS can use the tree to quickly search for existing rows with that value before allowing a row insertion (or column update) to proceed.
- **Primary key** is just a unique key, but there can only be one per table
  - We think of the primary key as the *most important* unique key in the table
- **Foreign key** makes a column's values match a column in another table
  - The referenced column in the other table should be indexed (usually it's the primary key).

Primary Keys

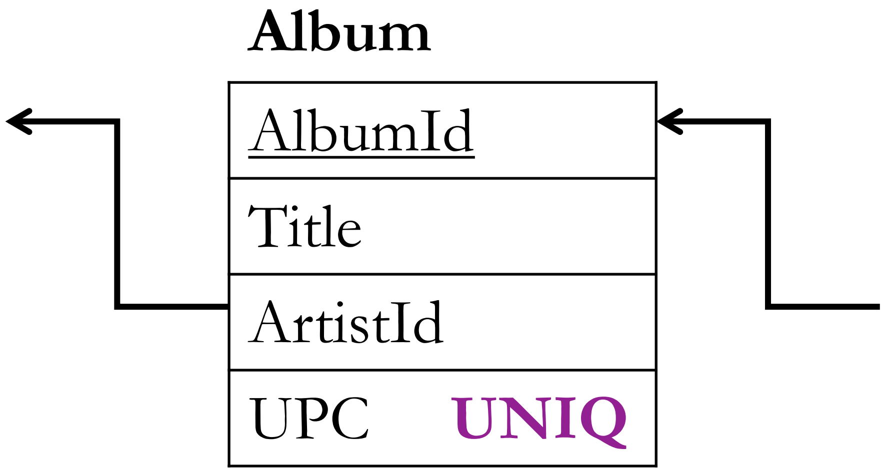
- Every table has a unique ***primary key*** – the column(s) that uniquely identify each row.
- No two rows can have the same primary key value.
- The primary key defines the principal feature of each row.
- Often it's an integer identifier
- **PlaylistTrack** table is different. It uses a *composite* primary key (made of two columns) and it lacks an integer identifier.

- In this class, we will **underline** primary keys in the diagrams.



Unique keys

- Unique keys are like additional (secondary) primary keys.
- No two rows can have the same value for a unique key.
- For example, we may wish to require that all Albums have both a unique AlbumId and a unique UPC (bar code):



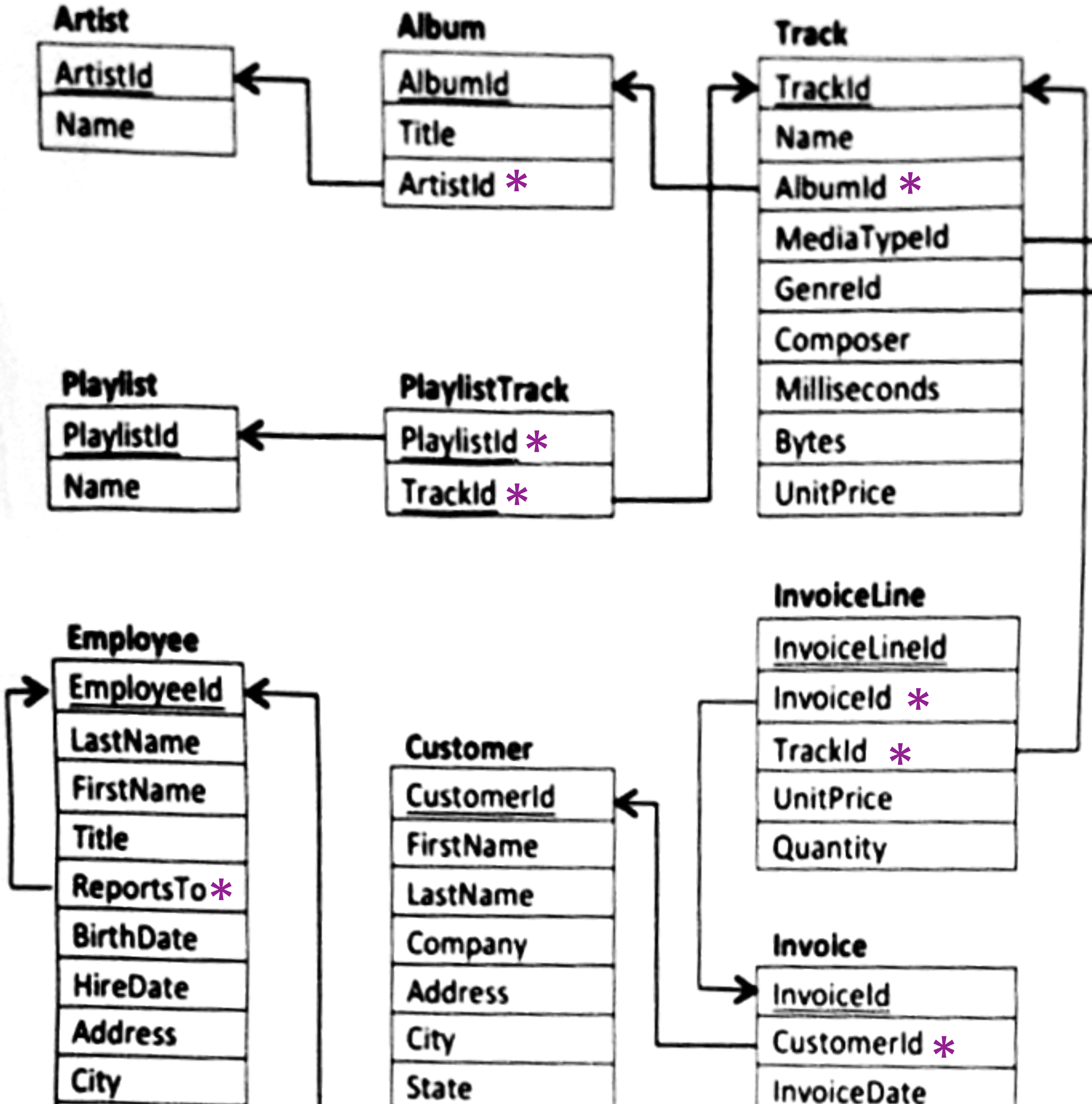
- We write **UNIQ** next to columns with unique keys in the diagrams

- When inserting data into this table, the new row must have both a unique AlbumId and a unique UPC.

Foreign Keys

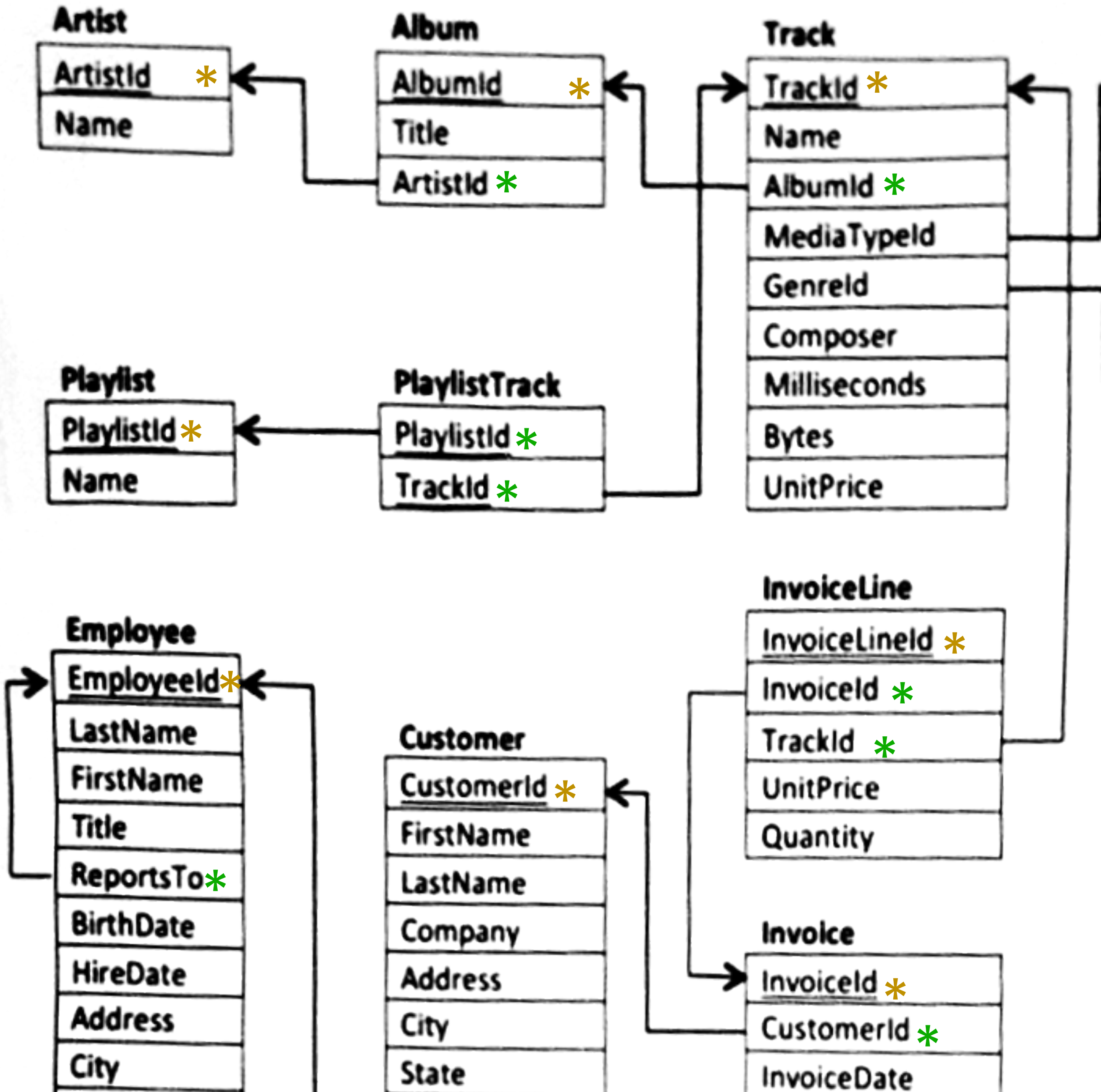
- Tables may be linked by *foreign keys* – columns that refer to keys in other tables.
- Usually these are integers ids, and should refer to a primary/unique key
- **PlaylistTrack** table is made entirely of foreign keys, so we call it a *linking table*.

- **Arrows** in these diagrams go from a foreign key to the column(s) they reference.



Parent and Child tables

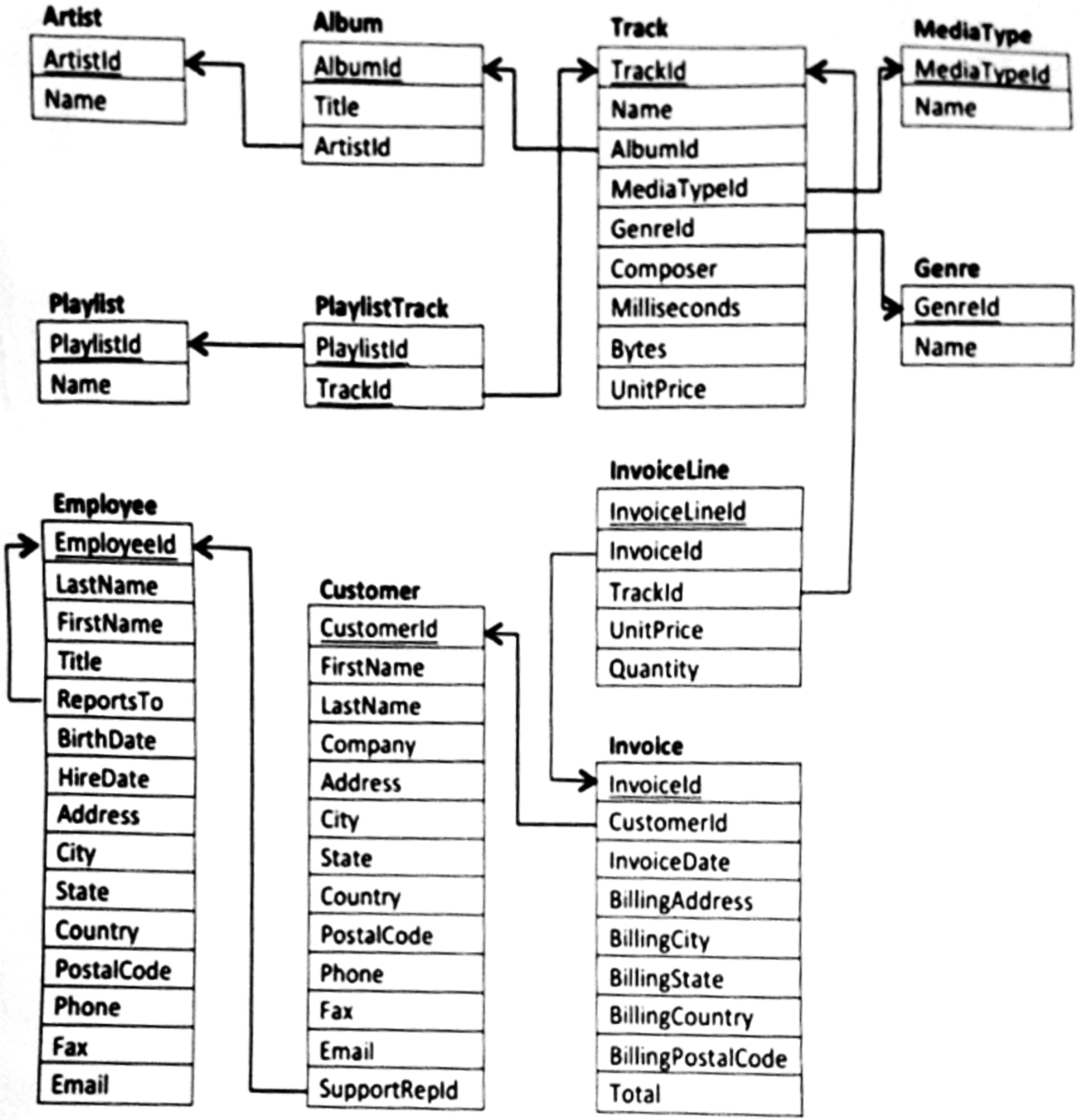
- Foreign keys define a parent and child table.
  - Child points back to parent
  - Parent row must be created before child row
- A table can simultaneously be both a parent and child.
  - Album is a child to Artist, but a parent to Track.



One to Many

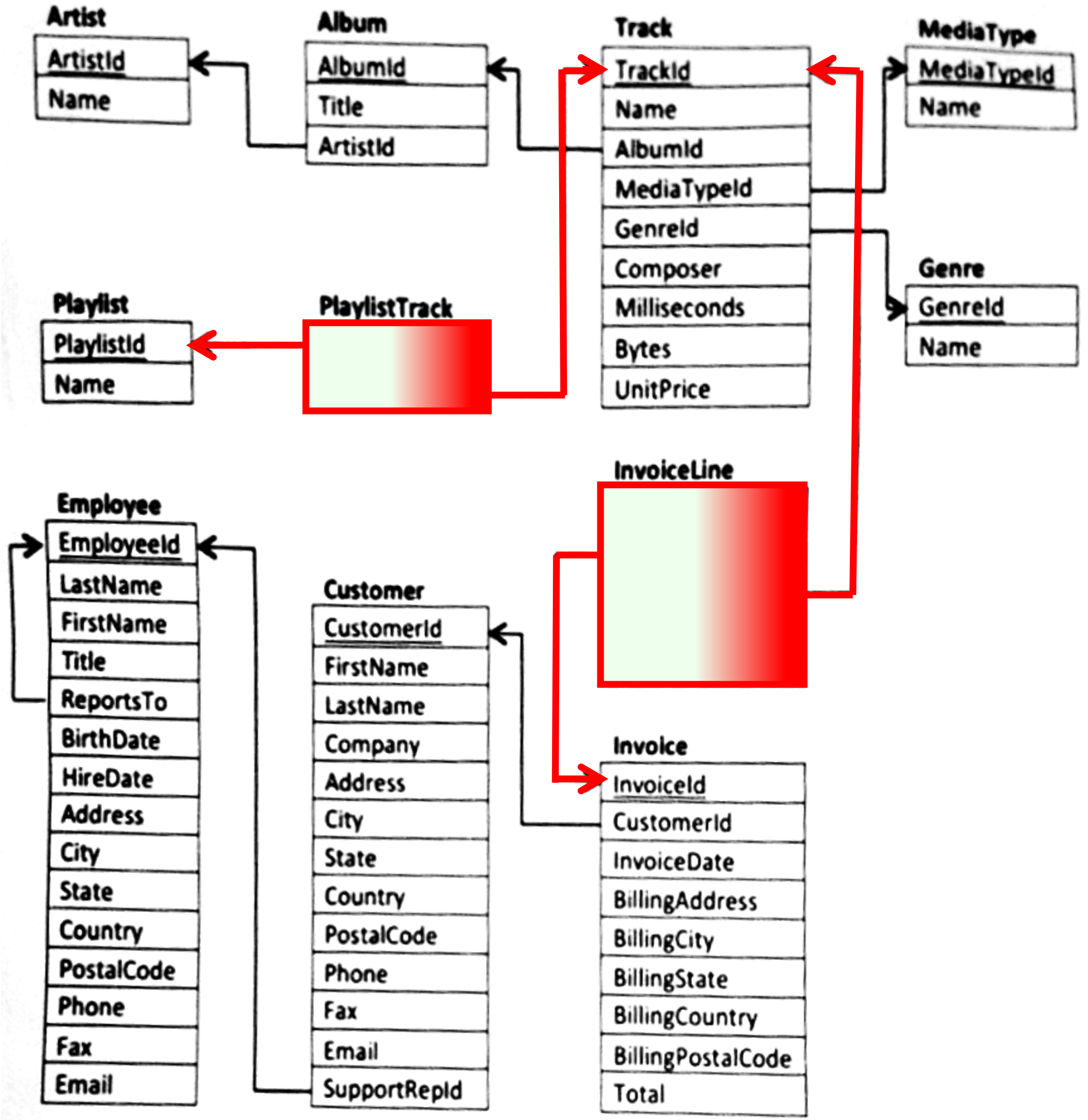
(or equivalently "many to one")

- Most foreign keys create one-to-many relationships
- Created when a column that is **not a primary key** has a foreign key.
- All of the arrows in this diagram represent one-to-many relationships.
  - Many of the rows in the child table can be related one row in the parent table.



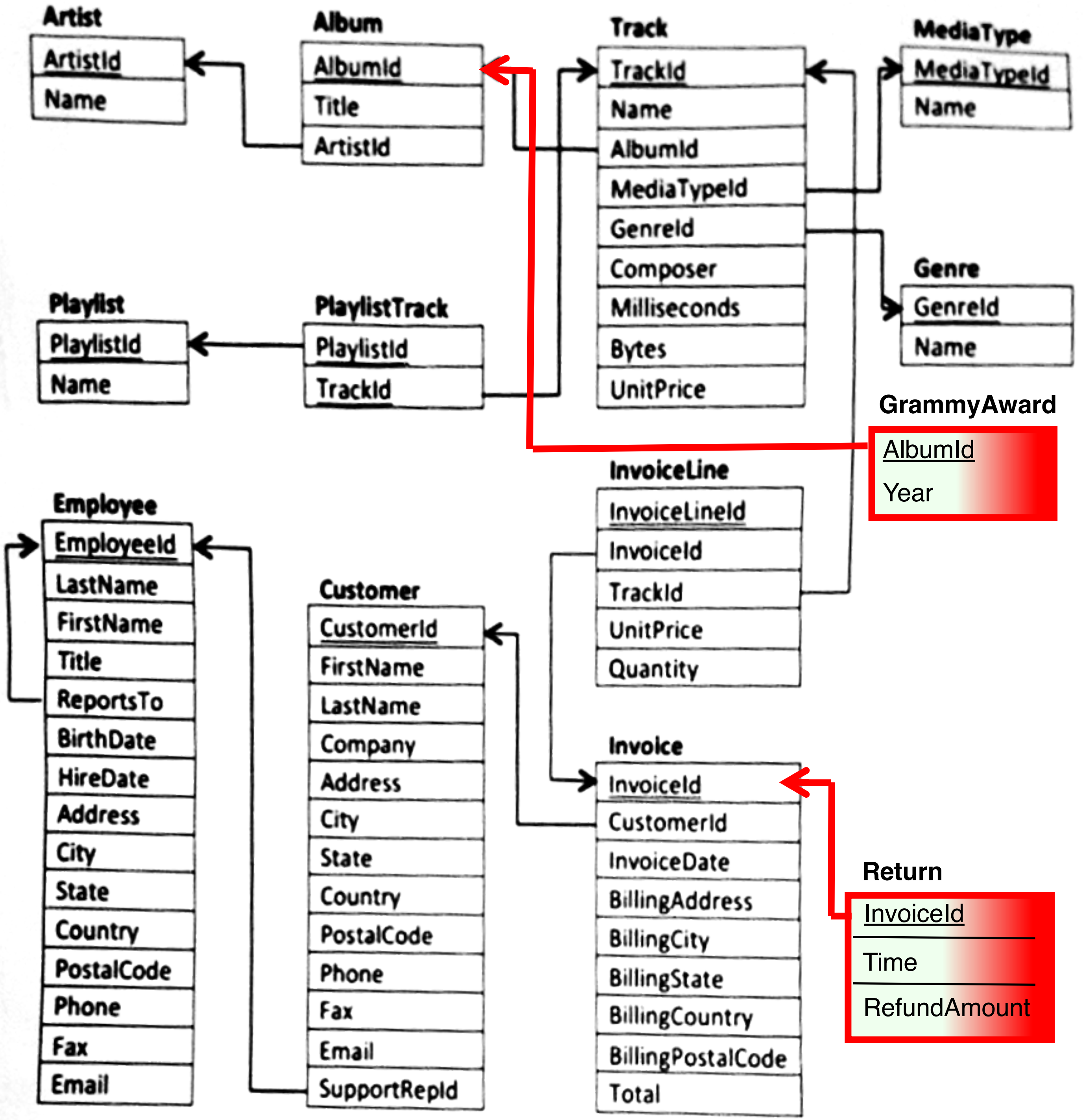
Many to Many

- Two one-to-many relationships starting at the same table can create a many-to-many relationship
- These are represented with ***linking tables.***
- But, some tables can be classified in multiple ways:
  - We think of **Track** as either an *object* or as a *many-to-many* relationship between albums and genres.



One to One

- One-to-one relationships exists when a primary (or unique) key is also a foreign key.
- In other words, there is an arrow pointing from one primary/unique key to another.
  - The fact that it's a unique key prevents it appearing multiple times (thus, not one-to-many).
- The child is a ***subset table***.
- Subset tables are an alternative to having optional columns in the parent table.



CREATE TABLE Syntax examples

from SchoolScheduling.sqlite

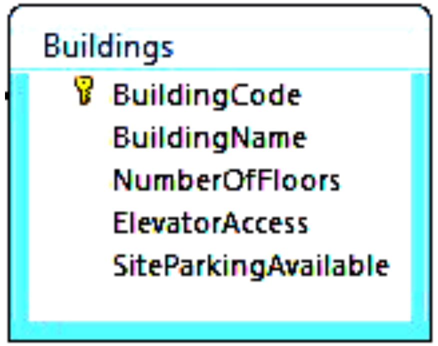


Table name

Required column, not optional

```
CREATE TABLE Buildings (
  BuildingCode nvarchar(3) NOT NULL,
  BuildingName nvarchar(25),
  NumberOfFloors smallint,
  ElevatorAccess bit NOT NULL DEFAULT 0,
  SiteParkingAvailable bit NOT NULL DEFAULT 0,
  PRIMARY KEY (BuildingCode)
);
```

Columns

Text with at most 25 characters

Column cannot be NULL, but it will take a value of zero if none is specified.

Each column has a data type, like nvarchar(3) *or* smallint

“Hello!” in ASCII

	H	e	l	l	o	!
hex	48	65	6C	6C	6F	21
binary	0100 1000	0110 0101	0110 1100	0110 1100	0110 1111	0010 0001

Variable length character encoding with UTF-8

1st byte	2nd byte	3rd byte	4th byte	# of free bits
`0... ....`				7 (ASCII)
`110. ....`	`10.. ....`			11
`1110 ....`	`10.. ....`	`10.. ....`		16
`1111 0...`	`10.. ....`	`10.. ....`	`10.. ....`	21

- Single-byte characters are identical to ASCII
- First byte tells you how many total bytes to expect
- Every “extra” byte starts with “10”
  - If you start reading in the middle of a character you’ll know it.
  - It’s very easy to know where each new character starts.

NBA_player_of_the_week.csv viewed as text

```
PlayerID,TeamID,PositionID,First Name,Last Name,Seasons in League,Height ,Weight,Age
1,20,7,Micheal,Richardson,6,77,189,29
2,14,9,Derek,Smith,2,78,205,23
3,9,2,Calvin,Natt,5,79,220,28
4,15,1,Kareem,Abdul-Jabbar,15,80,225,37
5,2,8,Larry,Bird,5,81,220,28
6,32,9,Darrell,Griffith,4,82,190,26
7,11,7,Sleepy,Floyd,2,83,170,24
8,8,8,Mark,Aguirre,3,84,232,25
9,15,7,Magic,Johnson,5,85,255,25
10,1,8,Dominique,Wilkins,2,86,200,25
11,33,6,Tom,McMillen,9,87,215,32
12,6,9,Michael,Jordan,0,88,215,22
13,7,4,World,Free,9,89,185,31
14,10,7,Isiah,Thomas,3,90,180,23
15,18,6,Terry,Cummings,2,92,220,23
16,6,6,Orlando,Woolridge,3,94,215,25
17,30,1,Jack,Sikma,7,95,230,29
18,22,8,Bernard,King,7,96,205,28
19,25,1,Moses,Malone,8,97,215,29
20,9,8,Alex,English,8,98,190,31
21,26,6,Larry,Nance,3,99,205,26
22,13,1,Herb,Williams,4,101,242,28
23,25,6,Charles,Barkley,1,102,252,23
24,32,8,Adrian,Dantley,9,85,208,30
25,18,9,Sidney,Moncrief,6,89,180,28
26,27,9,Clyde,Drexler,2,95,210,23
27,29,9,Alvin,Robertson,1,98,185,23
```

JSON

- JavaScript Object Notation
- Used in many web applications and data APIs
- Allows an arbitrary amount of **nesting**
- Spaces are ignored, except within quotes.

Basic components are:

- `[]` for ordered lists
  - Items are separated by commas
  - Items can be any JSON
- `{}` for unordered dictionaries/objects
  - Key: value pairs are separated by commas
  - Keys must be strings (text)
  - Values can be any JSON
- Numbers, `true, false, null`
- Strings (text) in double quotes `"..."`

```
[
  {
    "name": "John",
    "age": 30,
    "cars":
       ["Ford", "BMW", "Fiat"]
  },
  {
    "name": "Alicia",
    "age": 32,
    "hometown": "Seattle"
  }
]
```

XML

- e**X**tensible **M**arkup **L**anguage
- Older than JSON, and now is less common than JSON because many people think XML is unnecessarily complicated.
- HTML is an XML document that defines a web page.

Basic components are:

- Text
- Tags
  - **<tagname>…</tagname>** or just **<tagname>**
  - Have a name, and have XML inside
  - Each start tag has a corresponding end tag, but only if it has data inside.
- Attributes
  - `<tag attr="value" …>`
  - Appear within tags
  - Attribute name and value must be text
  - Tag can have multiple attributes, but each must have a unique name

```xml
<people>
  <person name="John"
          age="30">
    <cars>
      <car>Ford</car>
      <car>BMW</car>
      <car>Fiat</car>
    </cars>
  </person>
  <person name="Alicia"
          age="32">
    <hometown city="Seattle">
  </person>
</people>
```

Comparison of data exchange formats

	Proprietary	SQL	CSV	JSON	XML
Space efficiency	Compact binary representation	Bloated text with SQL syntax	Text with little extra syntax	Text with little extra syntax	Text with verbose tag names
Compatibility (readable by many)	Must use specific program/DB	Each DBMS has its own SQL dialect	Standardized format	Standardized format	Standardized format
Expressibility (data complexity)	Complex relationships	Complex relationships	Represents a single table	Complex relationships	Complex relationships
Popularity	Rare	Rare	Common	Common	Less common
Flexibility/rigidity	SQL DBs are have a clearly defined schema that must be obeyed.		Rows all have same columns.	Data and schema are defined together. Different elements can have different attributes.	

- Text-based file formats (SQL, CSV, JSON, XML) are not space efficient, but text files can be compressed using general-purpose file compression utilities like gzip to alleviate the problem (eg., `my_data.json.gz`)

Fixed point example in 16 bits

Let's store the chemical elements' atomic weights.

- Smallest value (hydrogen) is 1.00784
- Largest value (uranium) is 238.02891
- Negative values are not possible
- We can reserve 8 bits for the fractional part and 8 bits for the part > 1
- In this particular binary fixed point representation, weight of uranium is:

  The radix point is implicit, not stored in the computer.

  `11101110.00000111`

  $= 238\,\frac{7}{256} = 238.02734375$ (We had to round off, so this is not precisely accurate)
- And the weight of hydrogen is:

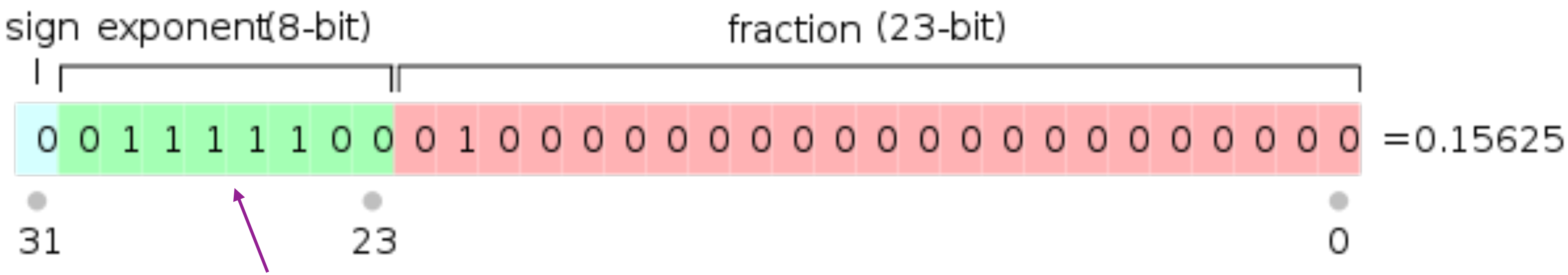
  `00000001.00000010`

  $= 1\,\frac{2}{256} = 1.0078125$

Representing floating point in bits

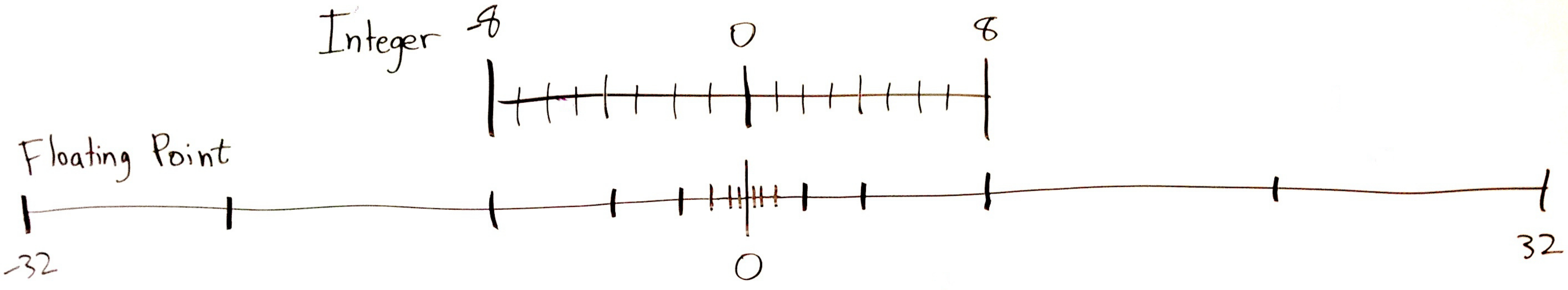
$$0.15625_{ten} = 0.00101_{two} = 1.\underline{01} \times 10^{-11}{}_{two}$$

- Three essential parts are the **sign**, **fraction**, & **exponent**
  - Notice that the first significant figure is always "1" so we don't have to store it
- In the mid 1980s, the IEEE standardized the floating point representation of 32 and 64 bit numbers:
  - The exponent has a sign too, but the standard says to add a "bias" of 127



1111100 = 124   124-127 = -3 exponent

Floats just distribute numbers differently



- Above, the dashes represent possible numbers.
- Both of the above number lines have 17 dashes (possible numbers)
- The only difference is the spacing.
    - Integer spacing is constant but floats are ***exponentially spaced***

Number Representation summary

- Computers represent numbers with different binary encodings
- **Text** can represent decimal numbers in various formats (eg., CSV, JSON).
- **Integers** represent whole numbers
  - Remember that $\mathbf{2^{10} = 1024} \approx 1000$, $\mathbf{2^{32} \approx 4}$ **billion**
  - Signed integers use two's complement
  - Used for *counting* and *identifying* records.
- **Fixed point** adds an implicit radix point to an integer.
  - Allows representing fractional quantities as integers, but with limited range.
  - Used for numbers that *should round off*, like prices.
- **Floating point** is a binary scientific notation representation
  - Can represent tiny fractional values and huge values with equal precision
    - Single precision ≈ 7 decimal digits, Double precision ≈ 16 decimal digits of precision
  - Used for *measurements* and *calculations.*

Bulk vs. online data sources

- So far, we have assumed that we can **bulk export** and **import** data.
  - In other words, we can get easily get all the data in one download.
  - Data is exchanged as CSV, JSON, XML, or SQL files:
    - **Dump** file(s) from origin database
    - **Load** file(s) into the destination database
- However, some data sources do not allow bulk access, and instead provide some kind of web-based access to the data:
  - A **data API** may be provided for users to query the data programmatically.
  - Data may be presented in web page for human reading, not intended for programmatic access.

Web scraping

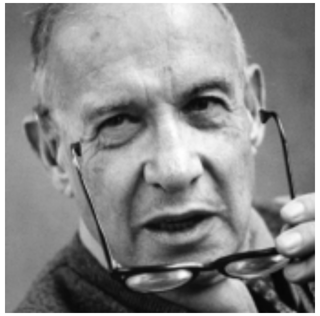
- Find the pages that hold the data
  - Often you'll start with a hard-coded index page and then programmatically look for links to additional pages.
  - Download the HTML (using Python **requests** package, for example)
- Extract the data from a given page:
  - Web pages are usually generated by a computer program, so the data will always be found within a certain pattern of HTML code.
- Locations in the HTML document can be specified in one of two ways:
  - **CSS selectors** – used be web page designers in Cascading Style Sheets to specify which fonts/colors/etc. *(styles)* apply to which parts of the page.
    - Python **beautifulsoup4** package uses CSS selectors
  - **XPath queries** – used for finding elements in an XML document (remember that HTML is a type of XML).
    - Python **lxml** package used XPath
  - CSS selector and XPath syntax can be tested in the Chrome developer tools.

Messy data

- Data can have missing, incorrect, or inconsistent values for many reasons:
  - Pulled from different sources with different naming or unit conventions
  - Paper scanning (OCR) errors
  - Human input errors
- Variety of tools are needed to deal with messy data:
  - Review summary statistics
  - Synonym tables
  - Named entity matching with ML (dedupe.io and Open Refine)
  - Crowdsourcing: MTurk, home-grown solutions
- Above all, don't blindly trust data you are given!

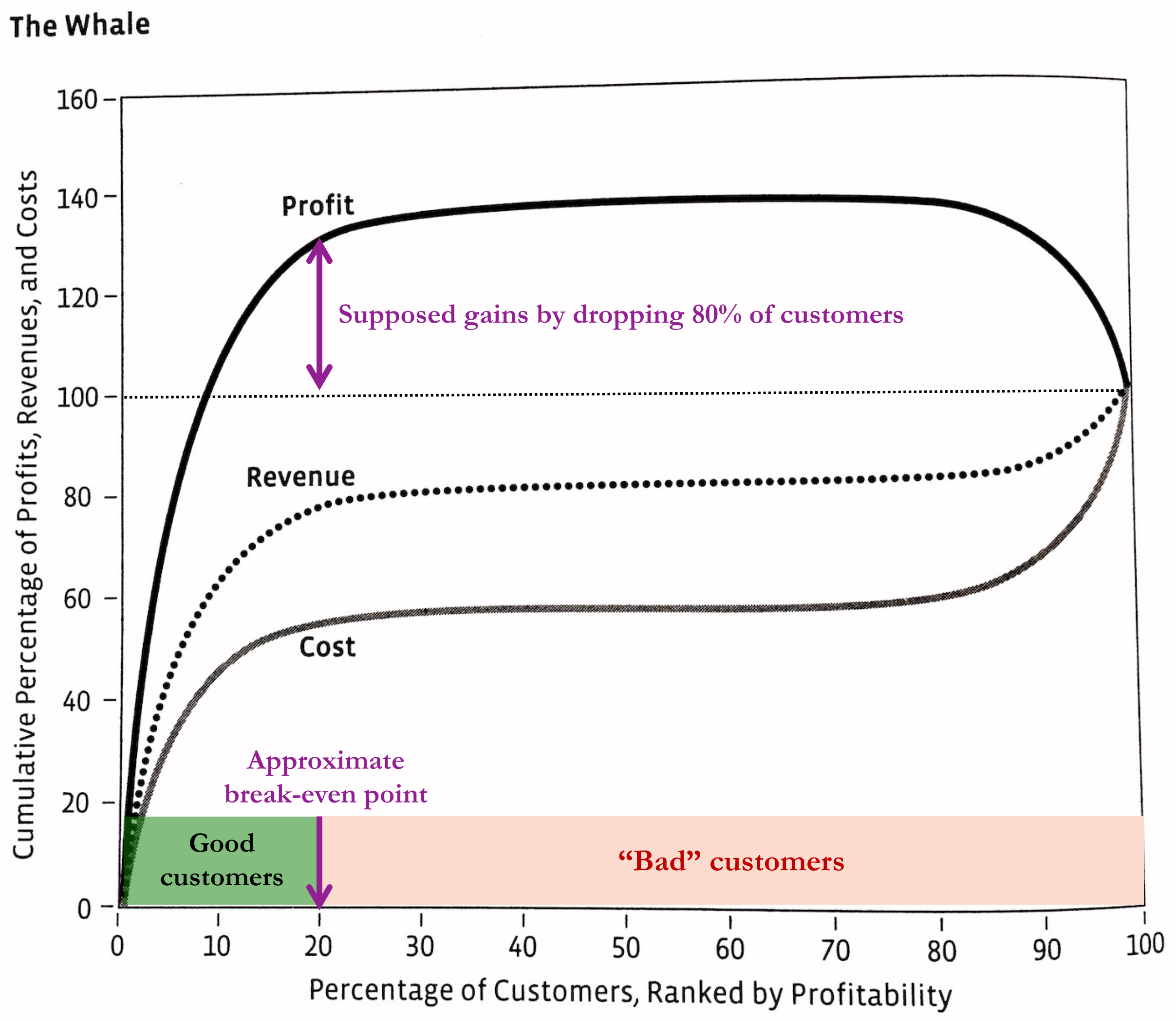
Data modeling practice

Final words of wisdom



- Peter Drucker famously said:
  - **“If you can’t measure it, you can’t improve it.”**
- MBA programs teach “scientific management” principles.
- Data-driven decision making is popular.
  - **However**, data can be easily cherry-picked and misinterpreted.
  - Focusing on the measurables is easy, but avoids important long-term issues.
  - Many of a business’ most important qualities **cannot be queried from the databases**:
    - Customer satisfaction.
    - Employee morale.
    - Brand image.
    - Long-term sustainability.
- Unfortunate reality:
  - **“If you can’t improve it, measure it!”**

Beware of "whaling" analysts



From "The Management Myth" by Matthew Stewart.

Naïve analysis:

- Profits could be increased by more than 30% by focusing on the top 20% most profitable customers!

Fatal assumptions:

- It's possible to drop the "bad" customers without also losing many "good" customers.
  - Market dominance has no effect on ability to attract the good customers.
- Per-customer profitability is constant over time.
  - Some of those "bad" customers may be very profitable next year (and vice versa).

Use data wisely

- Analysis doesn’t stop when you get a numeric “answer” or a plot.
- Ask yourself:
  - What’s missing from this analysis?
  - What are we not measuring (where we could search for more data)?
  - What cannot be measured?
  - Do the results change if we look at different time periods or random subsets of the data? (This indicates a lack of statistical significance.)
- Now ask the same questions above to the people most intimately familiar with the business.
- Data related to human activities are always a simplification of reality!
- View data as a **scientist** – keep testing your assumptions.