cs 143 notes

yash lala
<yashlala@gmail.com>

class info and intro: lecture 1

primarily application and usage based, rather than theoretical.

the relational model: defining data

data is represented in a tabular format.

tuples
: rows of the data.

attributes
: columns of data.

domain
: the set of possible values an attribute can take (equivalent to **typing** in a programming language).

relation
: a table.

schema
: the structure of relations in a table (eg. `Student(id, name)`).

instance
: an actual table that follows the schema (data).

keys
: set of attributes that uniquely identifies a tuple in a relation.

in the abstract model, the data uses set semantics; duplicate tuples not allowed, tuple and attribute order doesn't matter.

we need a null value to represent unknown/not applicable data. this complicates our DBMS, and gives us unintuitive answers at times. we need to use 3-valued logic to deal with database queries (true, false, unknown; concrete rules to deal with null and unknown variables). we'll go over this later.

sql is used as a DDL (data definition; eg. set a schema) and a DML (data modification lang). we'll learn the DDL part below.

SQL as a DDL

sql has data types.

- char(n): fixed length string. length is *always* `n`. shorter strings will be padded with underscores.
- varchar(n): variable length string, length up to `n`
- integer: 32 bit
- real and double: floating point, 32 and 64 bit respectively
- decimal(a, b): fixed point numbers! `a` := total number of digits in base 10, `b`:= how many of them are decimals. eg. `999.99` is of type decimal(5, 2). useful for money.
- date: eg. "2010-01-15"
- time: "13.50.00".
- timestamp: concatenation of date and time. stored as string.
- datetime: not part of the sql standard, but used in mysql. a more modern timestamp, maintains time zones, etc.

sql reserved keywords are case insensitive, we capitalize by convention (implementations are diff, so assume sensitive). created a database key with the given types below:

```
CREATE TABLE Course(
        dept CHAR(2) NOT NULL,
        course_number INT NOT NULL,
        section INT NOT NULL,
        instructor VARCHAR(50),
        PRIMARY KEY (dept, course_number, section),
        UNIQUE (dept, section, title)
);
```

we can set some data attributes as *keys*; they'll be unique to each tuple, and can be used to easily access the data. to specify that a field must never be null, we suffix the type with a `NOT NULL` statement.

some more statements:

- `CREATE TABLE` makes table, `DROP TABLE` deletes it.
- one primary key per table (they'll be unique). use `UNIQUE` to mark other attributes as unique.
- `DEFAULT` to set a default value for attribute.
- `LOAD DATA INFILE myfile.csv INTO TABLE mytable` will read data from a file into a table.

querying data: the relational algebra

relational algebra, formal language for querying data; returning relations from a relation. theoretical foundation behind SQL. uses set semantics; no duplicates, no order (SQL doesn't do this for performance reasons).

main operators:

$\sigma_C(R)$
: selection operator. filters out rows in a relation. `C` is a condition to filter on (eg. `name = 'yash'`), and `R` is the data to filter. instead of using `!=`, this traditionally writes `<>`.

$\pi_C(R)$
: projection operator. projects a relation into a smaller number of dimensions (ie. takes only some attributes from a dataset). eg. $\pi_{name}(Students)$ will get all of the student names.

$\times$
: cartesian product operator.

$\rho_{N(A_1, A_2)}$
: rename operator. renames a relation. `N` is the new name, and the `A` variables are the new attribute names. the parenthetical is optional.

$\bowtie$
: "natural join" operator. joins two tables on attributes that are the same (eg. given 2 tables with (id, name) and (id, age), return a new table with (id, name, age)). enforces equality on all common attributes. can be composed from cartesial product, select, and rename.

$\cup$
: union operator. just like sets. schemas should be the same (technically even in name, although we often ignore this for convenience. it's ok if their types are the same). removes duplicates.

$\cap$
: intersection operator. just like sets.

$-$
: set difference. sometimes notated as $\setminus$.

most of these can be constructed from other operators. we need selection, projection, cartesial product, renaming, union, and set difference.

general technique for relational algebra: when it's hard to formulate a query, think of its complement.

SQL

modelled on relational algebra. descriptive language; doesn't tell **how** to execute the query (that's left to the *relational database management system* (RDBS)).

queries generally look like

```
SELECT attribute1Iwant, attribute2Iwant
FROM table1IwantItFrom, table2IwantItFrom
WHERE conditionIwant;
```

we can name tables like `...FROM LongTableName T, LongerTable Z, ...`.

analogous to $\pi_{a_1, a_2}(\sigma_{c}(T_1 \times T_2))$. note that the "select" clause in the SQL is actually what to *project* on; the relational algebra select part is under the condition.

the selection conditions (aka the stuff in the `WHERE`) support the usual C-like logical operators, but `<>` replaces `!=`, and `AND` replaces `&&`. it even has a form of shell-style regex matching:

```
...something
WHERE addr LIKE '%Wilshire%'
```

here, `%` is a wildcard like the shell `*`, and `_` is equivalent to the shell `?`. we also have string processing functions such as `UPPER`, `LOWER`, `CONCAT`, etc.

multiset semantics; preserves duplicates. supports set operations, which *don't* preserve duplicates: `UNION`, `INTERSECT`, and `EXCEPT` (set difference; goes between two SQL queries). if we want duplicates for set operators, we should use `UNION ALL` etc. if we want no-duplicates for regular queries, we should say `SELECT DISTINCT`.

sql subqueries

SQL supports subqueries; select statements can appear inside other select statements; ie. select statements return relations (which we can use wherever we'd normally use a relation). there's a special case; if a result of a query is a 1-tuple 1-attribute relation (eg. name="steve"), then we can use it like a constant value. eg.

```sql
SELECT sid
FROM Student
WHERE addr=(SELECT addr FROM Student WHERE sid=301)
```

the above query will get the SIDs of all students living in the same address as the student with SID 301. easy peasy. by theorem, we can rewrite every subquery as a toplevel query as long as there's no negation (`NOT`). if there is a `NOT`, we'll need an `EXCEPT` (set difference) statement.

be careful! subqueries can be subtly different in terms of edge cases/ number of duplicates returned!

we have some set membership operators (`IN`, `NOT IN`). these work nicely with subqueries (put a tuple on the left and a table on the right). we also extend our comparisons operators: `a < ALL b`, `a >= SOME b`, etc. these do what they sound like; they'll be true only if a is larger than `ALL` values in table b, etc. `EXISTS()` returns true if its given a nonempty table.

subqueries can use variables from the outer queries. this effectively runs the query once for every "query match" of the outer query. these are called *correlated subqueries*:

```
SELECT name
FROM student S
WHERE EXISTS(SELECT * FROM Enroll E WHERE E.sid = S.sid)
```

conceptually: the outer query runs 1 tuple at a time, and binds the tuple to S. then for each S, we run the inner query and check the condition. of course, this may not happen under the hood; but we can visualize it like this.

we can have subqueries in the FROM; essentially aliases. very convenient for expressing same subquery multiple times. this is called a common table expression.

```
WITH alias AS (subquery)
SELECT ...
```

SQL aggregates and group by

we want to get information from more than one tuple; eg. sum, avg. this isn't a part of relational algebra, so we make SQL more expressive.

we can specify these "aggregate operators" in the `SELECT` clause. eg:

```
SELECT AVG(gpa)
FROM Student
WHERE sid IN (SELECT sid...)
```

this makes SQL more expressive than relational algebra; they extended relational algebra to add this capability.

be careful when combining these with duplicates (easy to make errors). when we need to do so, write like `SELECT AVG(DISTINCT gpa)`. the distinct qualifiers go inside.

what if we need to aggregate over subgroups? (eg. the avg. gpa of every "age" of student?). we use the `GROUP BY` clause.

```
SELECT age, AVG(gpa)
FROM Student
GROUP BY age;
```

this partitions the set into partitions over the "age" attribute (each age gets a different group). for every group, I apply the aggregate functions. group by might not always make sense; eg.

```
SELECT sid, age, AVG(gpa)
FROM Student
GROUP BY age;
```

the above query doesn't make much sense; student IDs are unique, so every "group" contains multiple SIDs. doesn't know which one to return. to solve this: when using group by, select can have only aggregate functions/attributes that have a single value for each group.

now the question is: what if we want to filter groups + aggregate functions? (eg. we've grouped by SID and counted the number of enrolled classes in the table; now how do we ignore all students taking 1 or less classes?). we could use subqueries; do the group by in one query, then apply another sql statement to the output of that query. this is inconvenient to type, so we have an extra block that's syntactic sugar for a subquery; the `HAVING` clause.

we use `HAVING` to filter elements during a group-by. eg: "find all students who take 2 or more classes":

```
SELECT sid
FROM Enroll
```

```
GROUP BY sid
HAVING COUNT(*) >= 2
```

the having clause is applied as a filter over each group.

having and where are often confused. I'll quote a useful passage from wikipedia.

```
A HAVING clause in SQL specifies that an SQL SELECT statement must
only
return rows where aggregate values meet the specified conditions.

HAVING and WHERE are often confused by beginners, but they serve
different purposes. WHERE is taken into account at an earlier stage
of a
query execution, filtering the rows read from the tables. If a
query
contains GROUP BY, data from the tables are grouped and aggregated.
After the aggregating operation, HAVING is applied, filtering out
the
rows that don't match the specified conditions. Therefore, WHERE
applies
to data read from tables, and HAVING should only apply to
aggregated
data, which are not known in the initial stage of a query.
```

in other words, we can read our SQL query in clause order (aside from the `SELECT` part, which is interpreted last) and we'll get an intuitive idea as to how it works internally. `WHERE` will filter the rows, `GROUP BY` will split and aggregate the rows, *then* `HAVING` will pick only certain groups.

sql insertion, modification, and deletion

insert some explicitly specified tuples: `INSERT INTO tablename VALUES (tuple1component1, tuple1component2), (tuple2component1, tuple2component1)`. we can also insert results of query: `INSERT INTO honors (SELECT * FROM students WHERE gpa > 3.7)`.

delete: `DELETE FROM relation WHERE condition`.

update: `UPDATE relation SET A1=V1, ...An=Vn WHERE condition`. eg. increase all CS course numbers by 100: `UPDATE class SET

SQL lecture 3

last time, we talked about aggregate functions and subqueries. aggregate functions are cool. they extended SQL beyond what's possible in relational algebra. now we'll go over some more extensions to SQL, that makes SQL more expressive than relational algebra.

- window functions
- order by

- limit

sql window functions

what if we want to return the name, GPA, and **overall avg** (in the grade) GPA of a student? this comes up in real-life cases when we want to figure out if data points are higher or lower than the average.

we might want something like `SELECT name, gpa, AVG(gpa)`; but this doesn't make sense. why? we know that aggregate functions and group-by are part of the same idea; when we don't specify a `WHERE`, the `AVG` is done over the entire set of input tuples (ie. treating them as a group). how can we return a single gpa for the group when everyone in the group has different GPAs? we talked about this before, and it's still a problem here. we don't want multiple input tuplies combined into a single output; we want **multiple** outputs for a single input (and the computation of some of the values may involve a more tuples). this is the idea of a window function.

use the `OVER` keyword; `SELECT name, gpa, AVG(gpa) OVER()`. one input tuple is generated per output tuple, but `AVG(gpa)` is computed over all. if we wanted the average of all students in the same age group, we could do `OVER(PARTITION BY age)`, which is exactly the same as group-by. it really should be the same syntax. now, the `AVG` function is only computed over a certain window.

sql `ORDER BY`

if we want to order the results, we can use `ORDER BY` clause near the end of our query. list attributes to sort by (will first sort by x, then by y), and specify `ASC` or `DESC` for ascending vs descending sort.

sql `FETCH FIRST` and `OFFSET` (aka unix `HEAD`).

we can also limit the results returned by `FETCH FIRST 3 ROWS ONLY`, or `OFFSET 4 ROWS FETCH FIRST 10 ROWS ONLY` (the second one returns rows 4-14). because queries can be fetched in any order, theoretically we should only be able to use this with an `ORDER BY` statement. in practice, most SQL implementations will just vomit out the first `n` random tuples anyways.

this was standardized very late (adding orders breaks relational algebra, so the purists faught against it), so a lot of SQL systems implement their own versions such as `LIMIT 3 OFFSET 2` (mySQL).

sql semantics recap

our total SQL syntax looks like:

```
SELECT attributes, aggregate1, aggregate2 OVER(PARTITION BY window)
FROM relations
WHERE conditions
GROUP BY attributes
HAVING aggregate conditions
```

```
ORDER BY attributes
FETCH FIRST n ROWS ONLY
```

it's interpreted in this order (except SELECT is interpreted last).

SQL lecture 3, part 2

now we've shown some non-relational-algebra parts of SQL. these non-relational extensions can lead to some tricky edge cases when learning SQL. we'll go over these tricky edge cases, and then go over some more non-algebra extensions.

- null values
- outer join
- multiset semantics for set operators
- brief discussion of SQL's expressive power, and recursion.

SQL: accounting for `NULL`

it's not necessarily intuitive what `NULL` should compare to in an operation. null is the absence of information; what should it evaluate to when it is compared with something?

the language designers chose to create a new system of 3-valued logic, `true`, `false`, and `unknown`. unknown represents when we can't determine a value. when we're returning tuples for a query, we **only** return the `true` statements, not the `false` or `unknown` ones. 2 interesting things about this:

1. this can lead to some unexpected results.
2. is `unknown` just the same thing as false? no! there are some cases where expressions with `unknown` will evaluate to true! (eg. `true OR unknown` will return true. think about it — makes total sense!).

Here's a truth table:

AND	true	false	unknown
true	true	false	unknown
false	false	false	false

OR	true	false	unknown
true	true	true	true
false	true	false	unknown

and if an input to an arithmetic operator is null, then its output is unknown.

makes sense so far. how should we handle null values in aggregates? this is tricky. if we're looking at averages, etc, then it intuitively makes sense that we should ignore null values. this isn't consistent with our previous logic! if we add null + 3, it's still null! ...but it's really practical (if we have a huge table with 1 null, we don't want the single null to screw up all of our data). so the decision was made: aggregate functions ignore nulls in most cases. this is wack! `SUM(gpa) / COUNT(gpa)` won't be `AVG(gpa)`!

some strange cases: `COUNT(attr)` won't count null values for that attr, but `COUNT(*)` **will** count the null values as well.

what if we run an aggregate on an empty table? `AVG` etc will all return null, but `COUNT` will return 0.

what about `NULL` and set operators? all set operators treat `NULL` as just another value.

also, because `NULL` is such a pain in the ass, we'll make a way to test if a value is null; `IS NULL` operator (we can't use `= NULL` because that's like we're comparing to a null value, which will always return unknown! tricky).

Note this is a very common mistake. make sure to remember it.

SQL: outer joins

say we have a student table an a class enrollment table. we want to return the number of classes every student takes; **including** students taking no classes (ie. absent from the enroll table). normally we'd just join the tables and `COUNT(class_id) GROUP BY student_id`; but this isn't enough here. if a student isn't enrolled in **any** classes, they're going to be absent from the enrollment table `(student_id, class_id)` entirely, and won't be in the joined result at all. we need to extend SQL to somehow "preserve" the entries in one table but not another when joining.

this is called the outer join! it preserves "dangling tuples". dangling tuples are tuples without a corresponding partner during a join. the question remains; what do we join the student with, if there's nothing in the enroll table? we fill it in with ``null``s. the outer join works like a regular join (`SELECT * FROM student, enroll WHERE student.sid = enroll.sid`); after looping through all entries, if there's an entry we haven't matched (ie. a sid in the student table that's not in the enroll table), we'll add a tuple: `(student.sid, NULL)`. now when we do `COUNT`, we can count this tuple for our total number of classes.

```
/* we don't do count(*) because that would count the null vals */
SELECT sid, COUNT(E.sid)
FROM Student AS S LEFT OUTER JOIN Enroll AS E ON S.sid = E.sid
GROUP BY S.sid;
```

the syntax is (in man page form): `FROM table1 [LEFT/RIGHT/FULL] OUTER JOIN table2 ON table1.attr = table2.attr`.

left outer joins add "dummy" entries for dangling tuples on the left, right joins do it for tuples on the right, and full joins do for both. here, if there's a student in enroll that's **not** in our student table, we want to ignore their entry. why are they there if they're not a student? we only care about students, after all. hence left outer join.

SQL: multiset semantics

I've time travelled back and edited the previous sections to add in this info. you already know `UNION ALL` etc. ;)

`UNION ALL` adds the cardinalities of elements in both sets, `INTERSECT ALL` yields the minimum cardinality, and `EXCEPT ALL` (set difference) subtracts. eg. `{a, a, a} EXCEPT ALL {a} = {a, a}`. its mostly intuitive.

leads to nonintuitive results; set operators are still commutative, but the distributive property no longer holds for SQL operators.

SQL: expressive power

we've added a lot of extensions to sql from relational algebra. it's not turing complete. we could make it turing complete very easily; allow for user definition of aggregate operators (`AVG` etc).

theorists didn't want SQL to be turing complete, so they nixed the revision to allow it to be so. they did, however, add a weaker version that gives us a lot more power (although not TC yet). this is recursive queries; we'll go over that next lecture.

SQL lecture 4, part 1: recursive SQL queries

this is the last SQL lecture. it covers a single extension that people added in 1999 to make SQL more expressive.

say we have a `Parent` table with 2 attributes: `(child, parent)`. how do we find all ancestors of susan? not possible with what we've learned so far; need to iterate queries **or** do something like

```
SELECT p1.parent, p2,parent, ...
FROM Parent AS p1, Parent AS p2, ...
WHERE p1.parent = p2.child AND p2.parent = p3.child...
```

we want to self join an unlimited number of times until we hit some condition. this is accomplished via something known as a **closure**. closures are implemented via recursion; it uses the same syntax as the common table expression (aka an alias):

```
/* make a table called "Ancestor".
 * it's going to map a person to every one of their ancestors
 * (aka. (susan, jim) means that jim is susan's ancestor).
 * lets define it recursively, by first finding everyone's *parents*
 * (via the parent table), then finding the parents' parents, and so on.
 */
WITH RECURSIVE Ancestor(child, ancestor) AS (
        /* non recursive part.
         * we want to store everyone's parents; this is just the Parent
         * table.

         * keep in mind; this isnt' really a "base case" or termination
         * condition like how they normally teach recursion. it's
         * really a "seed point"; the table will grow from here!
         * when will it stop? when we stop finding new ancestors (if you
         * think and use an example, you can convince yourself this will
         * always happen. you don't have to think that hard, either).
         */
        (SELECT * FROM Parent)

        /* now we union it with the recursive part */
        UNION

        /* let's add the parents of the parents to the list!
         * take the product of Ancestor and Parent (for loop).
         * recursion, so assume we've already got a list of ancestors.
         * the parent of your ancestor is also your ancestor, so we want
         * to add the ancestor's parents too!
         */
        (SELECT P.child, A.ancestor /* take the youngest and oldest */
        FROM Parent AS P, Ancestor AS A
        WHERE P.parent = A.child)) /* if they share the same middle */

        /* and because of recursion, we're done!
         * sql will keep evaluating the above table until it stops
         * growing (or otherwise hits a fixed point).
         * note that this isn't an infinite loop. think about it
         */

/* now let's be specific and find susan's ancestors. */
SELECT ancestor FROM Ancestor WHERE child='Susan';
```

it'll recursively go until the table reaches a fixed point (`f(x) = x`).

lecture 1/27: the entity-relationship (ER) model

we'll talk about how to draw table designs in diagrams.

the ER model

ER model is like UML. graphical representation of database information, can be converted to tables via tools. consists of entities (nodes) and relationships (edges).

entity classes (eg. students) are rectangular nodes, and have properties (represented as elliptical nodes) linked to them. if a property is a key, underline the text in the elliptical node.

entity classes can be in **relationships**; eg. students attend classes. in this case, there’s a line drawn from one rectangular node to another, and a "diamond" node in the middle of the line with the name of the relationship. by default, not all entities in the entity classes have to be in the relationships; to indicate that, make a double-line on the required side(s) of the diamond node (eg. all classes need to be taught by prof, not all profs teach class. the double line will be on the class side of the diamond).

relationships can include different numbers of entities from each side; they have **cardinalities** (one to one, one to many, etc). indicate this by drawing an arrowhead on the entity class on the "one" side of the relationship. I think of the arrow kinda like "certainty"; when you go along it, you’re sure which object you’ll end up on because there’s only one. you’re not sure the other way; in theory, there could be many other rectangles just like you that you don’t know of. we can also represent cardinality by a `1..3` notation; entity participates in 1-3 relationships. `*` represents any number of times.

sometimes you need more than binary relationships; eg. student has a class and a TA. if that’s the case, we can hook up multiple entity classes into the relationship diamond node.

sometimes its useful to write text above each edge. eg if students partner with each other, we’ll end up with a student rectangle linked twice to a "partner" relationship tag. we can differentiate it by writing "coder" and "tester" above the two lines that link it to show its role in the relationship.

sometimes we want to extend a relationship class; eg. domestic vs foreign students are both in student table, but foreign may need some more info (eg. country of origin, visa). we do this by "superclassing" or subclassing; draw a triangle coming from the base, and write `ISA` ("is a"; ie. foreignstudent is a student). draw edges from the other sides of the triangle to the subclasses. the subclasses inherit all behaviors and attribute circles from their parent. if every instance of the parent **needs** to be a subclass (total specialization eg. abstract class), then double-line the superclass edge, just like before.

sometimes, a database may not have a key. take a database of students and a database of project (project report, project number, etc). looking at the project report name or project report number, we wouldn’t automatically know which student it refers to, so the project data wouldn’t have a "key" on its own; it needs to be used in conjunction with the student table to get a unique identifier. in other words, the "key" to that database is really a foreign key that points to an element in the student table. these are **weak entity sets**, and they’re notated by using a double rectangle and double diamond (double diamond for the identifying relationship) in the ER diagram. the **discriminators** of a weak entityset are the attributes that would be a key (when combined with the main table information); for example the project number in the above table. these are notated by underlining the attribute ellipse.

converting these to tables is very straightforward; it’s exactly what you would intuitively do anyways. every diamond relationship gets a table that maps keys from the left to the right. weak entity-sets get tables with foreign keys to the main entity-set (ie. for projectreport, include studentid).

lecture 1/27 + 12/03: relational design theory

sometimes, attributes are entirely determined by other attributes (eg. student id → student name). say we have an enrollment table with the schema `(student id, student name, class)`. multiple enrollments will lead to storing the name multiple times. redundant data takes space and leads to complications; if we unenroll a student from their last class, we'll actually delete their info from our db. plus, takes up more space.

normalization theory strives to reduce redudancy by reducing the functional dependencies within a table. the resulting "good" table schemas are called *normal forms*. the most common normal form is boyce-codd normal form (BCNF). other normal forms (third normal, fourth normal) exist; third normal is mainly a curiosity, but fourth normal is commonly used.

functional dependencies and their implications

say we have a table R, where X and Y are sets of attributes in R (eg. `X = {name, sid}`. don't confuse yourself later — X is a set of columns, not tuples!)

`X -> Y` is a *functional dependency* if the map R from X to Y is a function (ie. `X1 = X2` logically implies `Y1 = Y2` where `Y_i = R(X_i)`). we sometimes call `X -> Y` a logical implication.

- if `Y \subseteq X`, the relation is fully trivial (always true). not useful.
- if `X \cap Y = \emptyset`, the relation is meaningful.
- if `X \cap Y \neq \emptyset`, some elements are trivial. we can remove the duplicate elements from the right hand side to convert to a nontrivial relation.

  transitive closure of a set of FDs(`F+`)
  : functional dependencies are transitive. `{A -> B, B -> C} \implies {A -> C}`. given a set F of FDs, we want to expand `F` fully to look at all the implied relationships. do this iteratively (find out all implications, add them to the set, find all implications, etc. iterate until we hit a fixed point; this is `F+`.).

  closure of a set of attributes (given ruleset F) (`X+`)
  : notation shorthand for "all attributes logically implied by attr set X given functional dependencies F".

when projecting to a smaller table, always remember to project `F+`, not `F`. dependencies can hide from you!

we can now formalize our notion of a database key.

key
: a set of attributes K is a key of a relation if:

  1. its closure is equal to the whole relation (ie. it implies everything else).
  2. no subset of K fulfills property 1.

keep in mind that this is a mathematical definition; it doesn't require keys be unique, because relations are sets (no duplicates) in mathworld. in SQLworld, we add the extra condition that keys should be unique in the table.

BCNF decomposition

split one table into 2 tablees to reduce intra-table FDs in the same table.

lossless join decomposition

a decomposition is a *lossless-join decomposition* iff natural joining the two tables produces exactly the original table equivalent condition: lossless-join iff *the shared attributes uniquely determine one of the decomposed tables* (ie. the shared attributes are keys for 1 table).

functional dependencies don't always lead to data duplication (keys determine all other values, but don't cause redundancy bc each key is only stored once). if the left hand side of the rule is a **key**, then there can be no redundancy. this is the idea between BCNF; if you think, you'll realize the tables will losslessly join, too.

boyce-codd normal form (bcnf)
: a relation R is in BCNF iff for every nontrivial functional dependency `X -> Y`, `X` contains a *key*.

conversion to BCNF is straightforward; recursively split table `R` into two until every subtable is in BCNF. to split: if you find a relation `X -> Y` \` that breaks the BCNF conditions, split into `R_1(X+)` and `R_2(X \cup Z)` where `Z := AllAttributes \setminus X+`.

know that BCNF composition may not be unique (eg. `(a, b, c)` where `a -> b`, `b -> c`. depending on which rule we pull out, we get `(a, b), (a, c)` vs `(b, c), (a, b)`.

> **Warning** whenever you decompose into BCNF, make **sure** to look at the closure of your FD set! rules hide behind other rules all the time!

splitting tables does reduce redundancy, but in practice decreases performance. as a rule of thumb irl, start with normalized tables, then merge them if the performance isn't good enough.

database integrity

databases may rely on the interactions between multiple tables. how can we ensure this data is "consistent"? we'll cover 3 things.

- referential integrity constraints
- check constraints
- sql triggers

first, some terminology.

integrity constraints are limitations on input data — if they're violated, they generate an error and abort the database operations. they take many forms:

- typed database fields are a form of constraints; a GPA is required to be a fixed point `decimal` number, for example.
- *key constraints*: define a set of attributes that should be unique in a table (ie. primary key / unique).

we also manage complexity with *database triggers*; they take actions based on certain events (defined via event-condition-action (ECA) rules).

now, go through the list in order.

referential integrity constraints

exactly what it sounds like. say we have an Enroll table that links students (via SID) to a class they're enrolled in (via `(course_num, dept, section)`). we want to make sure that these references are pointing to real students (ie. no dangling pointers), so we can explicitly ask SQL to treat them as pointers. SQL will then ensure that the references are valid before allowing insertion into the table. if we want an entry that doesn't point to a student or class, we insert the NULL value to disable pointer checking.

really, it's *exactly* like pointers. here, they're called **foreign keys**, because they key into another table. to create them, we use the syntax:

```
CREATE TABLE Enroll(
  sid INT,
  dept CHAR(2),
  cnum INT,
  sec INT,
  FOREIGN KEY (sid) REFERENCES Student(sid),
  FOREIGN KEY (dept, cnum, sec) REFERENCES Class(dept, cnum, sec));
```

the referenced attributes must be ``PRIMARY KEY``s or `UNIQUE` (otherwise the pointer would be ambiguous).

(if the key names are the same in both tables we don't have to specify them. this statement is also like the `PRIMARY KEY` statement; we can define it inline with the attribute if we want (`sid INT REFERENCES Student`).

it's useful to think about what operations can violate integrity.

- inserting/updating an entry into the Enroll table.
- deleting/updating an entry from the Student table.

by default, SQL will throw an error if constraints are violated. violations in the referencing (ie. enroll) table are never allowed, but we can define handlers for violating actions in the referenced table (ie. the Student table). say we delete a student; we can ask mySQL to delete its corresponding rows in the enroll table, which will maintain consistency. if we change a student's id, we can propagate (cascade) these changes into the enroll table too. there are two optional clauses:

```
CREATE TABLE E(
  a INT, b INT
  FOREIGN KEY (b) REFERENCES S(b)
  ON UPDATE CASCADE
  ON DELETE CASCADE);
```

in the place of cascade `CASCADE`, we can use:

- `CASCADE`: propagate changes to the referencing tables.

- `SET NULL`: set the referencing pointer to null.
- `SET DEFAULT`: change the referencing pointer to a default value. we'll have to define the default value beforehand.

> **Tip** be careful when using cascades! can cause huge data deletions, if there are circular references or self-referencing tables!

tables can reference themselves; a single attribute can reference another attribute in the table. this could be useful to define a graph, for example. cascades are extra scary here.

what if two tables reference each other? works as expected, but we need to declare them in a special way to avoid undefined table warnings (just like `let rec` in OCaml).

```
CREATE TABLE Chicken(cid INT PRIMARY KEY, eid INT);
CREATE TABLE Egg(eid INT PRIMARY KEY, cid INT REFERENCES Chicken);
ALTER TABLE ADD FOREIGN KEY (eid) REFERENCES Egg (eid);
```

insertion into an empty table is similarly annoying. we can either:

1. create the god of chicken, that came from nowhere (NULL) and have all other eggs stem from it.
2. create a chicken (and egg) that came from itself.

here's an example of approach 2.

```
INSERT INTO Chicken VALUES (1, NULL);
INSERT INTO Egg VALUES (1, 1);
UPDATE Chicken SET eid=1 WHERE eid IS NULL;
```

check constraints

allows more elaborate constraints on internal data. uses a condition similar to a where clause.

```
CREATE TABLE Student(
  sid INT,
  name VARCHAR(50),
  gpa REAL,
  CHECK (gpa >= 0 AND gpa <= 4)
);
```

the constraint can be complicated, and can include subqueries. it's specific to a particular table; whenever the table is updated, we reject the statement if the condition is violated.

> **Tip** when writing conditions, remember how boolean relationships are defined: "a implies b" can be written as `(not a) union b`.

here's an example: students with gpa less than 2 shouldn't take a CS class. aka: gpa less than 2 implies department is not equal to CS.

remember

```
CREATE TABLE Student(
  sid INT,
  gpa REAL,
  yadda yadda yadda
);
CREATE TABLE Enroll(
  sid INT,
  dept CHAR(2),
  cnum INT,
  section INT,
  CHECK (
    dept <> 'CS'
    OR
    sid NOT IN (SELECT sid FROM Student WHERE gpa < 2)
  )
);
```

remember, the constraint is evaluated only *when the Enroll table is updated*. check constraints aren't perfect; if a student's already in a CS class and *then* I update their gpa to 1, our check constraint won't trigger. this is also why we can't simulate referential constraints with check constraints; we need to keep track of errors on both the referrer and referred side.

database trigger

part of the SQL3 (1999) standard. like check constraints, but much more flexible. ECA (event-condition-action) pattern — for any event of that type, check the condition and then take the action.

general syntax:

```
CREATE TRIGGER TriggerName
<event>
  <referencing clause>  -- optional
WHEN (<condition>) -- optional
<action>;
```

`<event>` can be (`|` indicates choice, `[]` is optional): - `BEFORE | AFTER INSERT ON R` - `BEFORE | AFTER DELETE ON R` - `BEFORE | AFTER UPDATE [OF attr1, attr2...] ON R`

`<referencing clause>` can be: - `REFERENCING OLD|NEW TABLE|ROW as var, ...` - `FOR EACH ROW`: row level - `FOR EACH STATEMENT`: statement level

`<action>` any sql statement. multiple statements should be enclosed by `BEGIN` + `END`, and should be separated by semicolons.

eg. drop students from all classes when their GPA is below 2:

```
-- double hyphens are comments, by the way.

CREATE TRIGGER minGPA
-- the event to monitor:
AFTER UPDATE OF gpa ON Student

-- the below 2 clauses are optional.
REFERENCING NEW ROW AS updated_student
FOR EACH ROW

-- the condition to check:
WHEN (gpa < 2.0)
-- the actions to take.
-- begin and end are optional when we only have 1 statement.
BEGIN
  -- we can have as many statements as we want here.
  -- make sure to end them in a semicolon.
  DELETE FROM Enroll WHERE sid=updated_student.sid;
END;
```

we can ask for `OLD ROW` instead; this is useful when monitoring updates and deletes.

`FOR EACH ROW` specifies our rule as a *row level trigger*; if 5 rows are updated, we'll run this statement 5 times. we can also specify `FOR EACH STATEMENT`; this will only run the actions once (regardless of how many rows are changed). we won't be able to use `REFERENCING OLD/NEW ROW` in this case (as it's only run once per transaction): use `REFERENCING OLD TABLE` instead to name the changed elements in table format.

example 2: for every insertion to student, add a corresponding tuple to enroll. all students have to take CS 143!

```
CREATE TRIGGER Mytrigger
AFTER INSERT ON Student
REFERENCING NEW ROW AS new_student
FOR EACH ROW
BEGIN
  INSERT INTO Enroll VALUES (new_student.sid, 'CS', 143, 1);
END;
```

we can also do it table wise.

```
CREATE TRIGGER Mytrigger
AFTER INSERT ON Student
REFERENCING NEW TABLE AS new_students
FOR EACH STATEMENT
BEGIN
  INSERT INTO Enroll (SELECT sid, 'CS', 143, 1 FROM new_students);
END;
```

triggers can trigger other triggers — be very cautious around them. in addition, every implementation does triggers a bit differently. (mysql only supports referential integrity when using the InnoDB engine, `CHECK` constraints aren't in vanilla mySQL, `TRIGGER` can't update the same table, etc.). it's terrible.

Warning mySQL silently ignores constraints that it doesn't support! there are no warnings or errors; if it's supported by the SQL standard but isn't supported here, mySQL just won't add them. be very careful. always check that your constraints are enforced, and make sure to use the most conservative syntax (long form `FOREIGN KEY`, etc).

non-relational databases: mongoDB

web applications stored information in JSON (ie. a tree model). they needed a *persistence layer* to store their data; RDBMS wasn't a good fit for this (we can either store a json object as text in a single attribute, or we can try to convert the data into a format that works for tables. both of these are bad.

enter mongoDB. stores JSON data. rows are called *documents* (consist of BSON objects), and tables are called *collections* (a group of similar documents).

BSON is a binary repr of JSON; supports more datatypes, and is more compact. every BSON document in a collection needs its own unique `_id` field, it's basically a key.

mongoDB adopts javascript philosophy; very lassez faire.

- shell syntax is pretty much javascript.
- if you `use mydatabase;` and mydatabase doesn't exist, it'll create it for you. if you insert into `db.books`, it'll create a books collection.
- mongoDB doesn't require a database schema — one collection can store documents of any kind.
- when you drop the last table, the entire database disappears.

check out the videos + documentation for the syntax.

mongoDB has a more flexible attitude towards aggregates and more complicated filtering. the SQL select statement consists of a few clauses, which are all evaluated in order (`FROM`, then `WHERE`, then `GROUP BY`, then `HAVING`, etc). mongoDB aggregates ditch this ordering, and allow us to compose a single statement ("pipeline") out of many clauses ("stages").

do this by passing a list of JSON objects to `db.mydbname.aggregate()`. each object describes a particular stage; the data is passed through each of the filters (stages) in order, and is finally returned.

mongoDB preserves structure, allows nested objects + redundancy. however, restructuring + combining data is complicated and inefficient. relational databases "flatten" the data, and loses a lot of structure. that said, we can easily combine data with relational operators; rn, large complex queries are still best-served by the relational model.

mapreduce

we already know this from cs 134. transform tuples into `(key, value)` tuples; group tuples with the same key together, then reduce them into a single tuple.

hadoop is an open source implementation. apache spark (open source compute cluster software) also supports it.

databases and the disk

you know all this, you grew up in a house with a storage engineer.

magnetic disks: data is transferred in units of *blocks* to amortize high head seek time. `access time = seek time + rotational delay + transfer time`. seek time: how far the head has to move to the right track. rotational delay: how long the head has to wait on the track before the sector that we want shows up. transfer time: how long it takes to read the sector (geometry).

ssds also have a lot of overhead for random i/o! it's not just magnetic disks that have a *seek time*.

keeping in mind these characteristics, how to we store tables into disks efficiently?

if we know the size of a tuple, we can pack `n` tuples into a disk block. there's likely to be a remainder left over; if we leave it blank and put the next tuple on the next block, we call our storage **unspanned**. if we pack in the first few bytes of the next tuple, we call it **spanned** storage. unspanned storage is more common, just because it simplifies multiplying disk blocks (and disk is cheap). at worst, unspanned wastes just under 50% of disk.

what about variable length tuples? we can reserve the maximum space for the tuple (eg. `VARCHAR(30)`) on disk (*reserved space*). this wastes a huge amount of space. we can also store the tuples as *variable length*, and pack them tightly. this makes retrieving particular tuples difficult, and may cause problems if we want to expand a tuple later on. so instead, we can use a *slotted page* structure; it's basically an inode-style pointer system from 111). the first few bytes of a block are headers; they contain pointers to the beginnings of each tuple in the block. we don't want a superblock full of pointers in a different disk sector (do it block by block), because we'd have to write 2 blocks every time we update a tuple.

Note really, these problems are the same as we'd experience with regular file systems. go back to your 111 notes.

what about large binary/character tuples? (binary/character large objects; blobs and clobs). eg movie reviews. its usually worthwhile to treat these as separate objects; a tuple will contain the small attributes, and a pointer to large attribute data somewhere else.

sometimes, it's worth it to store data by column rather than by row. helps with large queries that only deal with some columns (plus, packing/compression is way easier). however, reading/writing rows is now very annoying.

how about the ordering of tuples within a block? initially, we could store them in key order (allows for efficient binary search lookup). however, tuple-packing becomes complicated when inserting data; either we can try to rearrange the existing entries to maintain this order, or we can store it as a linked list within the block (FAT style).

the slotted page structure is great for inserting new entries into a block (like a directory); but what if there's no more space in the block? we solve this with **overflow pages**; every block's header can point to another block that contains more data in that block (just like secondary inodes for full directories. this overflow data is a bit messy, so we don't typically fill blocks when creating tables (gives us room for more tuples later). oracle supports via `PCTFREE` (percentage free) SQL statement.

indexing data

we've discussed how to order/retrieve data at a fs level. now we do the same thing at a higher level via database indexing.

say we want to retrieve a student with a given student id. if we have 100,000 students in our database and 10 millisecond read time, this query will take about 170 milliseconds to run. that's not good enough (what if we have to look up a lot of students)?

we define an **index** on a table; an auxiliary data structure that helps us perform quick tuple lookups given a search key.

first, some quick categorization of indices.

terminology

say tuples are sorted according to a particular attribute. an index into this attr is the **primary index**. linear data can only have 1 order, so only one primary index can exist per table. also known as **clustering index** (consecutive/duplicate data blocks can be read in a cluster).

if tuples are unsorted (eg. we're looking up on a non-key attr), then we call our index a **secondary index** (or **nonclustering index**).

if there's one entry for every table entry, then we call this a **dense index**. if we only list the first tuple in a block, it's called a **sparse index**. note that we need primary clustering to set up a sparse index, because all consecutive values are consecutive (ie. in a cluster.). otherwise we'd need a dense index.

1. *what's the difference between clustering and nonclustering index?*

   same as primary and secondary. primary index is the actual order of data. secondary is when the data is unordered. remember; we need primary clustering to set up a sparse index.

indexed sequential access methods (ISAM)

indexes are sorted lists: `(key attr, pointer to full tuple)`; the attr to index on (ie. attr we can search by) coupled with pointers to the full tuple's location on disk. to search for a tuple, we binary search through the index.

to look for a given student id, we can binary search our index to get the current pointer, then go to the full tuple. why is this better? we can't generally do better than binary search; but with an index, the keys are stored much closer together (no other attrs stored) on disk, so fetching a single index block will fetch more entries to binary search on. this means less disk i/o; we can potentially cache the entire dense index in main memory (especially if the tuples are indexed by something small, like an integer).

we can do even better by using a **sparse index**. in our index, only store one `(key, pointer)` pair per block (pointing to the first tuple). now, we binary search for a block (eg. `83`); follow the pointer of the appropriate table entry (eg. `80 -> 0xlkj`), then look in the disk block to see if our tuple is there. this adds 1 read to our time (we need to check the actual disk block every time to see if our tuple is there), but greatly decreases our index size.

if even this isn't fast enough, we can have a *multi-level* index; binary search in the second-level index, this will give you a pointer to a first-level index. search in the first level index (which is often dense), and you'll get a pointer to the disk block with your data. exactly the same as multilevel page tables.

works exactly the same if we want to search an attr that's not the primary key (eg. student name). we call it **secondary index** (aka **nonclustering index**) this time, because underlying data isn't sorted (clustered) on the attr.

again: any first-level secondary index will have to be dense, because consecutive tuples have random values (unsorted). the second-level index onwards can be sparse.

recap

let's make sure we know all the keywords:

search key
: the attribute we search our "index" (auxiliary data structure for). can be different than table primary key.

primary index vs secondary index
: primary index points to sorted data, secondary index can't rely on sort.

clustering index vs nonclustering index
: same as primary vs secondary index.

dense index vs sparse index
: dense index contains an entry for every node, sparse index doesn't. only primary indexes can be sparse.

B+ tree indexing

insertion into ISAM indexes is bad; overflow tables have many problems. after a lot of insertions and deletions, the table structure degrades. B+ tree is a data structure that avoids most of these problems; self-balanced binary tree. guarantees that at least half of the memory used by the B+ tree is occupied.

> **Note** I've only made paper notes for this part. use them for the final.

extendible hash table indexing

hash-table equivalent to a B+ tree. constant time insertion/deletion from tree.

> **Note** I’ve only made paper notes for this part. use them for the final. this degree is a joke.

query optimization

join: cost model predicts the *best* join alg given data characteristics. we’re going to use "# of disk blocks read/written" as our metric. we ignore the last I/O for writing the final result, because that’s the same for all.

Last updated 2022-08-28 19:18:07 PDT