CS143: Database Systems

Professor Cho

Thilan Tran

Winter 2021

Contents

CS143: Database Systems

- a **database management system (DBMS)** is a way to manage and store data:
  - how does a database differ from a spreadsheet software like Excel?
    - expected to efficiently scale to a massive amount of data, without suffering
    - expected to persist the data
    - expected to provide secure and safe access to data
    - expected to be conveniently used by a large number of clients at a time
  - program data is not assumed to entirely reside in main memory RAM, but instead in disk
    - this leads to utilization of different data structures
- database architecture:
  1. disk for data (sometimes stored in main memory, if data can fit)
  2. OS
  3. DBMS engine
     - database system may access disk through OS, or directly through raw disk IO
  4. API
     - eg. standard APIs like JDBC (Java), ODBC (Microsoft)
  5. app or CLI
  - downloading a DBMS software like MySQL installs parts 3, 4, and a CLI
- popular DBMS software:
  - relational:
    - open source: MySQL, PostgreSQL
    - closed source: Oracle, Microsoft SQL, IBM DB2
  - non-relational (NoSQL)
    - MongoDB, Spark
- five steps in database construction:
  1. domain analysis
     - captured in an entity-relationship (ER) model or using unified modeling language (UML)
  2. database design
     - normalization theory
  3. table creation
     - using a data definition language (DDL)
  4. load
     - using SQL or bulk load
  5. query and update
     - using a data manipulation language (DML)

Data Models

- what is a data model? why do we need it?
  - eg. the human brain acts as a data model
    - remembers information in a way that the data is easily accessible and understandable
  - how do we translate the memory of a human brain to computers?
    - computers speak in binary and only store data in a very specific, concrete format
  - a **data model** is a very specific way to model or represent data in computers
- types of data models include graph, tree, relational models:
  - a **graph model** (or network model) has nodes, edges, and labels
    - eg. the most natural way to represent airline flights
    - initially the most popular data model
    - more flexible and semistructured, often used for JSON data eg. MongoDB
  - a **tree model** (or hierarchical model) is a graph arranged as a tree, eg. company hierarchies
  - in a **relational model**, all data is instead represented as a set of tables:
    - introduced in 1970 by Codd, and completely revolutionized the database field
    - in graph and tree models, accessing and modifying data is not as efficient:
      - instead, represent all data through relationships and store them in tables, like a spreadsheet, for more efficient access
      - a downside to the relational model is the rigidity of its schemas
    - in mathematics, tables are expressed as **relations** eg. tuples, triplets, etc.
    - now, the relational model is the most popular data model used in DBs

Relational Models

- relational model terminology:
  - **relations** ie. tables contain many **tuples** ie. rows, each with various **attributes** ie. columns
  - each attribute has a **domain** ie. type
  - a **schema** is the structure of relations in a database:
    - includes properties like the relation name, attribute names,

domains
    - eg. `Student(sid: int, name: str, addr: Addr, age: int, GPA: int)`
  - the **instance** is the actual data populating a relation that conforms to some schema
    - ie. schema is the variable type, and the instance is the variable value
  - **keys** are a set of attributes that *uniquely* identify a tuple in a relation:
    - multiple keys are possible
    - eg. in `Course(dept, cnum, sec, unit, instructor, title)` , the keys may be the department, course number, and section number
      - alternatively, department, section, and title if course numbers are repeated
    - generally, assuming the set of attributes for keys is the minimum set
      - in the worst case, with no duplicates, a relation's keys are all its attributes
  - more specifically, different types of keys:
    1. a **super key** is any key
    2. a **candidate key** is a key with the minimum number of attributes
    3. a **primary key** is a candidate key chosen to be used as the main key for a relation
       - shouldn't have null values
  - **null values** are used to indicate not applicable, uninitialized values, etc.:
    - are compatible with every type, unless otherwise explicitly defined
    - leads to complications, eg. in comparisons for conditional queries
      - thus DBMS may return unexpected answers
    - requires **3-valued logic** where every condition can be true, false, or unknown:
      - need concrete rules to deal with null and unknown values
      - adds increased implementation and execution complexity
- name scopes of relational models:
  - the name of a relation is unique across relations
  - the names of attributes are unique in a table
    - thought there cane be the same attribute names in different tables
- set semantics in relational models:
  - in a set, there are no duplicate elements, and order is unimportant
  - thus in the relational model:
    1. no duplicate tuples are allowed
       - whereas SQL allows duplicate tuples for practical reasons
    2. attribute order does not matter

Relational Algebra

- relational query langauges include:
  - formal languages eg. relational algebra, relational calculus, datalog
  - practical languages eg. SQL, Quel, QBE
- in relational algebra:
  - inputs and outputs are both relations, so **piping** is possible
  - set semantics are followed, so duplicates are automatically eliminated

Table 1: Students

sid	name	addr	age	GPA
301	John	183 Westwood	19	2.1
303	Elaine	301 Wilshire	17	3.9
401	James	183 Westwood	17	3.5
208	Esther	421 Wilshire	20	3.1

Table 2: Classes

dept	cnum	sec	unit	title	instructor
CS	112	01	03	Modeling	Dick Muntz
CS	143	01	04	DB Systems	John Cho
EE	143	01	03	Signals	Dick Muntz
ME	183	02	05	Mechanics	Susan Tracey

Table 3: Enrollments

sid	dept	cnum	sec
301	CS	112	01
301	CS	143	01
303	EE	143	01
303	CS	112	01
401	CS	112	01

- queries can be done using the formal relational algebra language
  - consider the following example queries using Table 1, Table 2, and Table 3

1. Get all the students:

$$Student$$

- to get all the tuples from a relation, just write the name of the relation

2. Get all students with age > 18:

$$\sigma_{age<18}(Student)$$

- the select operator $\sigma_C(R)$ filters based on boolean expression C
  - R can either be a relation or a result from another operator

3. Get all students with GPA > 3.7 and age < 18:

$$\sigma_{GPA>3.7 \wedge age<18}(Student)$$

4. Get student ID and GPA of all students:

$$\Pi_{sid,\text{GPA}}(Student)$$

- need a different operator
- the project operator $\Pi_A(R)$ filters column-wide based on attributes A
  - returns a new set of *columns*

5. Get all departments offering a class:

$$\Pi_{dept}(Class)$$

- due to set semantics, does not return two elements for "CS"

6. Get student ID and GPA of students with age < 18:

$$\Pi_{sid,\text{GPA}}(\sigma_{age<18}(Student))$$

- composing operators
- note that it is generally not useful to compose projections, since they can be reduced into a single projection

- the **cross product** or Cartesian product operator in relational algebra:
  - $R \times S = \{t \mid t = (r, s) \;\; \forall \;\; r \in R, s \in S\}$ concatenates every tuple of each relation together:
    - if $|R| = r$ and $|S| = s$, $|R \times S| = rs$
    - eg. $R \times S$ contains $a_1 b_2, a_1 b_2, \ldots, a_n b_m$
  - ie. creates one output per every pair of input tuples
  - useful when combining tuples from multiple relations

7. Get the names of students who take CS classes:

$$\Pi_{name}(\sigma_{dept=\text{'CS'}}(\sigma_{Student.sid=Enroll.sid}(Student \times Enroll)))$$

$$\Pi_{name}(\sigma_{dept=\text{'CS'} \wedge Student.sid=Enroll.sid}(Student \times Enroll))$$

- but this cross product is quite expensive, so we can change the ordering to reduce the size of the large cross product

$$\Pi_{name}(\sigma_{Student.sid=Enroll.sid}(Student \times \sigma_{dept=\text{'CS'}}(Enroll)))$$

- equivalently, using the natural join operator:

$$\Pi_{name}(\sigma_{dept=\text{'CS'}}(Student \bowtie Enroll))$$

- the **natural join** operator $\bowtie$ is used to join two tables *naturally*:
  - filters the Cartesian product by the tuples based on the equality conditions of *all* shared attributes
  - equivalent to:

$$R \bowtie S = \sigma_{\langle equal\ \ shared\ \ attributes\rangle}(R \times S)$$

Table 4: $Class \bowtie Enroll$

dept	cnum	sec	unit	title	ins	sid	dept	cnum	sec
CS	112	01	03	Modeling	Dick Muntz	301	CS	112	01
CS	112	01	03	Modeling	Dick Muntz	303	CS	112	01
CS	112	01	03	Modeling	Dick Muntz	401	CS	112	01
CS	143	01	03	DB Systems	John Cho	301	CS	143	01
EE	143	01	03	Signals	Dick Muntz	303	EE	143	01

8. Get the names of students who take classes offered by `'Dick Muntz'`:

$$\Pi_{name}(Student \bowtie (Enroll \bowtie \sigma_{inst=\text{'Dick Muntz'}}(Class)))$$

9. Get the names of student pairs who live at the same address:

$$\Pi_{S_1.name,S_2.name}(\sigma_{C_1 \wedge C_2}(\rho_{S_1} Student \times \rho_{S_2}(Student)))$$

   - where $C_1 = (S_1.addr = S_2.addr)$ and $C_2 = S_1.sid > S_2.sid$
   - crossing a relation with itself is also known as **self-join**:
     - need to use the rename operator $\rho_{S(A)}$
     - renames table to S and optionally, specific attribute to A
   - the comparison check on `sid` prevents duplicates such as
     - `('John', 'John'), ('John', 'James'), ('James', 'John')`

10. Get all students and instructor names:

$$\Pi_{name}(Student) \cup \rho_{Person(name)}(\Pi_{inst}(Class))$$

   - when using union, the schema must be equal between the relations
   - ie. the attribute names have to match up
   - no duplicate tuples in the result

11. Get all the courses (department, number, section) that no one takes:

$$\Pi_{dept,cnum,sec}(Class) - \Pi_{dept,cnum,sec}(Enroll)$$

  - easier to express this query using its complement
  - using the set difference operator $R - S$

12. Get instructor names who teach both CS and EE courses:

$$\Pi_{inst}(\sigma_{dept=\text{'CS'}}(Class)) \cap \Pi_{isnt}(\sigma_{dept=\text{'EE'}}(Class))$$

  - when using intersection, the schema should again be the same
  - $R \cap S = R - (R - S)$

13. Get IDs of students who did not take any CS class:

$$\Pi_{sid}(Student) - \Pi_{sid}(\sigma_{dept=\text{'CS'}}(Enroll))$$

- the core relational operators are:
  - $\sigma, \Pi, \times, \cup, \rho, -$
  - set difference is the only non-monotonic operator, so it is core
    - ie. when we add more input tuples, the output can *decrease* in size, unlike every other operator
  - set union is core because it is the only operator that *stacks* tuples on top of one another
    - cross product can only stack tuples *horizontally*
  - while the other operators $\bowtie, \cap$ can be expressed with other operators

SQL

- **structured query language (SQL)** is the standard langauge for interacting with relational DBMS (RDBMS):
  - many versions of the SQL standard exists, SQL92 or SQL2 is the main standard
  - has multiple components:
    - the **data definition language (DDL)** includes schema definitions, constraints, etc.
      - eg. `CREATE, ALTER, DROP`
    - the **data manipulation language (DML)** includes queries, modifications, etc.
      - eg. `SELECT, INSERT, UPDATE`
    - other components for transactions and authorization

Data Definition Language (DDL)

- the DDL component of SQL allows for expressing schema definitions
- basic common SQL data types:
  - string:
    - `Char(n)` with padded fixed length n
      - will pad shorter values
    - `Varchar(n)` with variable length and max length n
  - number:
    - `Integer` 32-bit
    - `Decimal(d, f)` with d total digits and f precision
      - eg. the max value of `Decimal(5, 2)` is `999.99` , useful for financial values
    - `Real` 32-bit, `Double` 64-bit
  - datetime:
    - `Date` eg. `2010-01-15`
      - no timezone in SQL standard!
    - `Time` eg. `13:50:00`
    - `Timestamp` eg. `2010-01-15 13:50:00`
      - on MySQL, `Datetime` is preferred instead
- SQL table creation:
  - `CREATE TABLE` statement

  - one `PRIMARY KEY` per table (but can be composed of multiple attributes):
    * `UNIQUE` is used for other keys (attributes may be null in SQL92)
    * by SQL standard, primary keys cannot be null
      · system will automatically mark those keys as not null, though the standard requires it to be explicitly written
  - `DEFAULT` sets the default value for an attribute
  - `DROP TABLE` statement for deleting a table

SQL table creation example:

```
CREATE TABLE Course(
    dept CHAR(2) NOT NULL DEFAULT 'CS',
    cnum INTEGER NOT NULL,
    sec  INTEGER NOT NULL,
    unit INT,
    instructor VARCHAR(50),
    title      VARCHAR(100),
    PRIMARY KEY(dept, cnum, sec),
    UNIQUE(dept, sec, title)
);
```

Loading Data

- there is no SQL standard for bulk data loading
  - in Oracle and MySQL, `LOAD DATA INFILE <file> INTO TABLE <table>`
- options:
  - comma vs. tab separation for columns
    * `FIELDS TERMINATED BY ','`
  - columns enclosed with quotes
    * `OPTIONALLY ENCLOSED BY '"'`

Database Manipulation Language (DML)

- SQL gives a high-level description of what a user wants:
  - given a SQL query, DBMS figures out how best to execute it *automatically*
  - the core query operators are selection, projection, and join (SPJ)

The following is the full general format for SQL `SELECT` :

```
SELECT attributes, aggregates
FROM relations
WHERE conditions
GROUP BY attributes
HAVING aggregate condition
ORDER BY attributes
FETCH FIRST n ROWS ONLY
```

- note that `SELECT` appears first, but is the last clause to be run
  - all other clauses can be semantically interpreted as executed in order

Relational Algebra Analogs

- consider the following example queries using Table 1, Table 2, and Table 3

1. Get the titles and instructors of all CS classes:

```
SELECT title, instructor -- project in SELECT clause
FROM Class               -- specify relations in FROM clause
WHERE dept='CS';         -- filter through condition in WHERE clause
```

- thus, the SQL statement `SELECT A1...An FROM R1...Rm WHERE C` is roughly equivalent to:
  - $\Pi_{A_1 \dots A_n}(\sigma_C(R_1 \times \cdots \times R_m))$
  - differences:
    * `SELECT` is projection rather than selection
    * SQL *does not* remove duplicates, uses **multiset** ie. bag semantics instead
      · sets can have duplicate elements and $\{a, b, a\} \neq \{a, b\}$
- some multiset semantics:
  - $R \cup S = S \cup R$ and $R \cap S = S \cap R$
  - but $R \cap (S \cap T) \neq (R \cap S) \cup (R \cap T)$
    * unlike in typical equivalence relation semantics

2. Get the names and GPAs of all students who take CS classes:

```
SELECT name, GPA AS grade -- renaming attributes, AS optional
FROM Student S, Enroll E  -- renaming operator for tuples, ie. tuple variables
WHERE dept='CS' AND S.sid=E.sid;

-- this returns duplicates if a student takes multiple classes!
SELECT DISTINCT name, GPA -- DISTINCT removes duplicates on final projected output
...
```

3. Get all student names and GPAs who live on Wilshire:

```
SELECT name, GPA
FROM Student
WHERE address LIKE '%Wilshire%';
```

- string variables:
  - `%` for any length string, `_` for a single character
  - eg. `LIKE %Wilshire%` matches any string containing `Wilshire`
  - eg. `LIKE ___%` matches any string with length ≥ 3
- other string functions include `UPPER(), LOWER(), CONCAT()`

4. Get all student and instructor names:

```
(SELECT name
FROM Student)
UNION -- set operator automatically takes care of duplicate instructors
(SELECT instructor -- can optionally rename to match input schemas
FROM Class);
```

- set operators:
  - eg. `UNION, INTERSECT, EXCEPT`
  - these operators *do* follow set semantics and remove duplicates
  - schemas of input relations should be the same
    * in practice, compatible types are fine
- to keep duplicates use `UNION ALL` etc. to enable bag semantics:
  - $\{a,a,b\} \cup \{a,b,c\} = \{a,a,a,b,b,c\}$
  - $\{a,a,a,b,c\} \cap \{a,a,c\} = \{a,a,b\}$
  - $\{a,a,b,b\} - \{a,b,b,c\} = \{a\}$

5. Get all SIDs of students who do not take any cs class

```
(SELECT sid FROM Student)
EXCEPT
(SELECT sid FROM Enroll WHERE dept='CS');
```

Subqueries

- SQL has extensions that allow certain queries that cannot be expressed using purely relational algebra, eg. subqueries and aggregate queries
- a SQL **subquery** is a nested `SELECT` statement within another:
  - the result from the inner `SELECT` is treated like a regular relation
  - if the result is a single-attribute, single-tuple relation, the result can also be used as a constant value ie. a **scalar-valued subquery**

- consider the following example queries using Table 1, Table 2, and Table 3

1. Get SIDs of students who live sith student 301:

```
SELECT sid
FROM Student
WHERE addr=(SELECT addr FROM Student WHERE sid=301)
      AND sid<>301;

-- without using subqueries:
SELECT S2.sid
FROM Student S1, Student S2
WHERE S1.addr=S2.addr AND S1.sid<>S2.sid AND S1.sid=301;
```

- using a scalar-valuled subquery
  - cannot always guarantee that the subquery is a scalar, but filtering on the primary key does guarantees the result is a single tuple
- if we can always rewrite subqueries in a non-subquery system, the two would be expressively equivalent:
  - generally, we *can* rewrite subqueries to non-subqueries as long as there is no negation
    - with negation, need to use `EXCEPT`
  - thus does not lend extra expressive power to SQL
    - ie. is a syntactic sugar for SQL

2. Get student names who take CS classes:

```
SELECT name
FROM Student
WHERE sid IN (SELECT sid FROM Enroll WHERE dept='CS');

-- without using subqueries:
SELECT DISTINCT name -- without DISTINCT, would return duplicates
FROM Student S, Enroll E
WHERE S.sid=E.sid AND dept='CS';
```

- the `IN` keyword is the set membership operator ie. $a \in S$:
  - the set may be a multiset, but `IN` would still not return duplicates

3. Get student names who take no CS classes:

```
SELECT name
FROM Student
WHERE sid NOT IN (SELECT sid FROM Enroll WHERE dept='CS');

-- without using subqueries:
```

```
(SELECT name FROM Student)
EXCEPT
(SELECT name
FROM Student S, Enroll E
WHERE S.sid=E.sid AND dept='CS');
```

4. Get student IDs who have higher GPA than any student of age 18 or less:

```
SELECT sid
FROM Student
WHERE GPA>ALL(SELECT GPA FROM Student WHERE age≤18);
```

- `ALL, SOME` are set comparison operators that compares a value against an entire set of values:
  - eg. `a>ALL R` , `a≤SOME R`
  - note that `=SOME` is equivalent to `IN` and `<>ALL` is equivalent to `NOT IN`

5. Get student names who take any class:

```
SELECT name
FROM Student
WHERE sid IN (SELECT sid FROM Enroll);

-- using a correlated subquery to reference an outer relation:
SELECT name
FROM Student S
WHERE EXISTS(SELECT * FROM Enroll E WHERE E.sid=S.sid);
```

- conceptually, for **correlated subqueries** to reference outer relations:
  - outer query looks at one tuple at a time and binds the tuple to `S`
  - for each `S` we execute the inner query and check the condition
  - real DBMS executes it more efficiently
- `EXISTS Q` is true if query `Q` returns one or more tuples
- subqueries can also appear inside a `FROM` statement, but they must be renamed
  - eg. `... FROM (SELECT name, age FROM Student) S ...`
- a **common table expression** can be used to alias a subquery
  - convenient for using the result of the same subquery multiple times

Using aliases:

```
WITH S AS (SELECT name, age FROM Student)
SELECT name FROM S WHERE AGE>17;
```

Aggregates

- an **aggregate function** is a *new* mechanism to combine information from multiple input tuples into a *single* output tuple:
  - rather than showing information from only a single input tuple per output
  - eg. `AVG, SUM, COUNT, MIN, MAX`
  - thus aggregates *do* increase the expressiveness of SQL compared to relational algebra
  - because the output is a single tuple, all selected attributes alongside an aggregate must be unique over the *entire* group
    - the groups selected to be aggregated can be adjusted using the `GROUP BY` operator
- consider the following example queries using Table 1, Table 2, and Table 3

1. Get average GPA of all students:

```
SELECT AVG(GPA)
FROM Student;
```

2. Get number of students taking CS classes:

```
SELECT COUNT(DISTINCT sid) -- need to avoid duplicates
-- SELECT DISTINCT COUNT(sid) is incorrect!
FROM Enroll
WHERE dept='CS';
```

3. Get average GPA of students who take CS classes:

```
-- incorrect, averages duplicate GPAs
SELECT AVG(GPA)
FROM Enroll E, Student S
WHERE E.sid=S.sid AND dept='cs';

-- subqueries are useful for handling duplicates
SELECT AVG(GPA)
FROM Student
WHERE sid IN (SELECT sid FROM Enroll WHERE dept='cs');
```

4. Get average GPA for each age group:

```sql
SELECT age, AVG(GPA)
-- SELECT sid, age, AVG(GPA) fails because an age group does not have the same sid
FROM Student
GROUP BY age;
```

- the `GROUP BY` operator partitions the groups that the aggregate function performs on
  - with `GROUP BY`, `SELECT` can have select aggregate functions or attributes that have a *single* value in each group

5. Get number of classes each student takes:

```sql
SELECT sid, COUNT(*)
FROM Enroll
GROUP BY sid;
-- this skips students who take no classes!
-- need to perform a union with those students, or use another technique
```

6. Get students who take two or more classes:

```sql
SELECT sid
FROM Enroll
-- WHERE COUNT(*)≥2 fails, aggregate cannot appear as part of WHERE
GROUP BY sid
HAVING COUNT(*)≥2;
```

- the `HAVING` clause is used to filter based on the condition of an aggregate
  - appears *after* `GROUP BY`

7. Using `HAVING` to get names of employees whose total salary is greater than all employees in Los Angeles:

```sql
SELECT name
FROM Work
GROUP BY name
HAVING SUM(salary) > ALL(
    SELECT SUM(salary)
    FROM Work W, Employee E
    WHERE W.name=E.name AND city='Los Angeles'
    GROUP BY W.name);

-- alternatively, using NOT EXISTS
SELECT name
FROM (SELECT name, SUM(salary) total-salary
      FROM Work
      GROUP BY name) Salary
WHERE NOT EXISTS(
    SELECT W.name
    FROM Work W, Employee E
    WHERE W.name=E.name AND city='Los Angeles'
    GROUP BY W.name
    HAVING SUM(salary) ≥ Salary.total-salary);
```

More SQL Extensions

- consider the following example queries using Table 1, Table 2, and Table 3

1. For each student, return their name, GPA, and the *overall* GPA average:

```
SELECT name, GPA, AVG(GPA) FROM Student;
-- is an error, using attribute names in an aggregate
-- that are not unique over the *entire* group

SELECT name, GPA, AVG(GPA) FROM Student GROUP BY sid;
-- does not give an error, but AVG(GPA) is the incorrect value

SELECT name, GPA, AVG(GPA) OVER() FROM Student;
```

- the `OVER()` function is the SQL **window function**, used after some aggregate `FUNCTION(attr)`
  - generates one output tuple per input tuple, but the function is computed over *all* input tuples

2. For each student, return their name, GPA, and the average GPA in their age group:

```
SELECT name, GPA, AVG(GPA) OVER(PARTITION BY AGE) FROM Student;
```

- the `PARTITION` clause applies the window function over a specific grouping:
  - analogous to `GROUP BY` but for window functions
  - ie. changing the window over which the parent aggregate function is computed

3. Order students by GPA:

```
ORDER BY GPA DESC, sid ASC; -- primary ordering GPA, secondary ordering SID
```

- since SQL is based on multiset semantics, tuple order is ignored:

    - but for presentation purposes, the `ORDER BY` clause orders the result tuples by certain attributes
    - the default ordering direction is `ASC` if omitted
    - note that this not change SQL semantics and is purely for presentation

4. Get top 3 students order by GPA:

```
SELECT * FROM Students
ORDER BY GPA DESC
FETCH FIRST 3 ROWS ONLY;
```

- can limit the number of returned tuples with the `FETCH` clause:
    - `[OFFSET <num> ROWS] FETCH FIRST <count> ROWS ONLY`
    - skip the first `num` tuples and return the subsequent `count` rows
    - this was standardized too late, MySQL variation is `LIMIT <count> OFFSET <num>`
- note that with all of its expressiveness, SQL is *not* a Turing-complete language:
    - in order to make SQL TC, a relatively simple extension is required
        * just have to allow for user-defined aggregate functions
    - many requests to add more expressiveness to SQL

Table 5: Example Database with Ancestry Relationship

child	parent
Susan	John
John	James
James	Elaine
...	...

5. Find all ancestors of Susan from Table 5.

```
SELECT parent FROM Parent WHERE child='Susan'; -- get parents

SELECT P2.parent grandparent
FROM Parent P1, Parent P2
WHERE P1.parent=P2.child AND P1.child='Susan'; -- get grandparents

...

WITH RECURSIVE Ancestor(child, ancestor) AS (
    (SELECT * FROM Parent) -- base case, initially populates Ancestor relation
                           -- with all parents
UNION -- UNION prevents duplicates
```

```
    (SELECT P.child, A.ancestor -- can now recursively use the Ancestor relation
     FROM Parent P, Ancestor A
     WHERE P.parent=A.child))
SELECT ancestor FROM Ancestor WHERE child='Susan';
```

- in order to find *all* ancestors, we need an additional **recursion** mechanism in SQL:
  - want to join multiple times until there are no more tuples to return
    - ie. running until a *fixed* point has been reached
  - can be structured as a graph problem eg. find all reachable cities given connections
  - support for recursion was added with SQL99, allows for computing closures of a set
    - when writing recursive queries, typically a `UNION` is used to connect a base ie. seed case with the recursive case
  - note that this recursive extension is not enough to make SQL TC

Handling `NULL`

What will be returned from the following query if GPA for some tuples is `NULL` ?

```
SELECT name FROM Student WHERE GPA * 100/4 > 90
```

- SQL is based on **three-valued logic**:
  - all conditions are evaluated to be `True, False, Unknown`
  - if input to an arithmetic operator is `NULL` , its output is `NULL`
  - arithmetic comparison with `NULL` then returns `Unknown`
    - eg. `NULL > 90`
  - SQL returns a tuple only if the result from condition is `True`
  - note that `NOT Unknown` is still `Unknown`
- assume GPA is `NULL` and age is 17:
  - `GPA > 3.7 AND age > 18` gives `Unknown AND False` which evaluates to `False`
    - `AND` short-circuits
  - `GPA > 3.7 OR age > 18` gives `Unknown OR False` which evaluates to `Unknown`

Table 6: Truth Table for Three-Valued Logic

`AND, OR`	True	False	Unknown
True	True, True	False, True	Unknown, True

`AND, OR`	True	False	Unknown
False	False, True	False, False	False, Unknown

Table 7: Example Database for Aggregates

sid	GPA
1	3.0
2	3.6
3	2.4
4	`NULL`

- handling null values in aggregates from Table 7:
  - SQL aggregates ignore null values and apply the aggregate on the remaining tuples
  - `SELECT SUM(GPA) FROM Student` gives 9.0
    * in theory `Unknown` would be more valid
  - `SELECT AVG(GPA) FROM Student` gives 3.0
  - `SELECT COUNT(GPA) FROM Student` gives 3
  - `SELECT COUNT(*) FROM Student` gives 4
    * `COUNT(*)` is the exception that also counts tuples that might have null values everywhere
  - when an input to an aggregate function is *empty*:
    * `COUNT` returns 0
    * all other aggregates return `NULL`
- handling null values in set operators:
  - eg. $\{2.4, 3.0, null\} \cup \{3.6, null\} = \{2.4, 3.0, 3.6, null\}$
  - `NULL` is treated like regular values for set operators
  - to checking `NULL` , can use `IS NULL` or `IS NOT NULL` :
    * note that `=NULL` and `<>NULL` do *not* work to check `NULL`
    * `=NULL` evaluates to `Unknown`

Get the number of classes each student takes, including 0-class students:

```sql
SELECT sid, COUNT(*)
FROM Enroll
GROUP BY sid;
-- this does not return 0-class students!

SELECT sid, COUNT(*)
FROM Student S, Enroll E
WHERE S.sid=E.sid
```

```sql
GROUP BY sid;
-- still does not return 0-class students!

SELECT sid, COUNT(cnum) -- CANNOT use COUNT(*) since it counts the null cnum as 1
FROM Student S LEFT OUTER JOIN Enroll E ON S.sid=E.sid
GROUP BY sid;
```

- in this example, students taking no classes become **dangling tuple**
  - these tuples do not get preserved in the join condition
- need to use the **outer join** function to preserve dangling tuples:
  - `<relation1> <dir> OUTER JOIN <relation2> ON <condition>`
    - can perform a `LEFT OUTER JOIN`, `RIGHT OUTER JOIN`, `FULL OUTER JOIN`
  - remaining attributes that are supposed to come from the other side are set as `NULL`

Data Modification

- the `INSERT` statement inserts a new tuple:
  - `INSERT INTO <relation> <tuples>`
  - `VALUES` keyword is used to literally specify tuples
- the `DELETE` statement removes tuples:
  - `DELETE FROM <relation> WHERE <condition>`
- the `UPDATE` statement updates tuples attributes:
  - `UPDATE <relation> SET <a1> = <v1>, ... WHERE <condition>`

1. Insert new tuples into the `Enroll` table:

```sql
INSERT INTO Enroll VALUES (301,'CS',201,1), (420,'EE',401,2);
```

2. Populate `Honors` table with students of GPA > 3.7:

```sql
INSERT INTO Honors (SELECT * FROM Student WHERE GPA>3.7);
```

3. Delete all students who are not taking classes:

```sql
DELETE FROM Student
WHERE sid NOT IN (SELECT sid FROM ENROLL);
```

4. Increase all CS course numbers by 100:

```sql
UPDATE Class
SET cnum=cnum+100
WHERE dept='CS';
```

Database Design

- how should we design tables in our database?
  - tables are not given
  - *"good"* tables may not be easy to come up with
- different database models:
  - **entity-relationship (ER)** model
    - developed alongside relational databases
  - **universal modeling language (UML)** is more generalized
    - supports less specialized tools compared to ER

Entity-Relationship Model

- the ER model is a graphical and informal representation of information in the database:
  - used to capture what we learn from domain experts as well as database users
    - **domain experts** clearly understand the relationships and nuances of the data we are trying to capture in a model
  - not directly implemented by DBMS
    - instead, tools exist to automatically convert E/R model into tables
  - has two main components, entity sets and relationship sets
- the **entity set** is a set of entities:
  - an **entity** is any thing or object in the real world
    - analogous to an instance of a class in OOP
  - entity set is analogous to a class in OOP
    - denoted by a rectangle in the ER model
  - an **attribute** is a property ie. field of entities:
    - denoted by ellipses in ER and connected to the entity
    - entities with attributes are like tuples
  - a **key** is a set of attributes that uniquely identifies an entity in an entity set:
    - all entity sets in ER need a key
    - denoted by underlined attributes in ER
- the **relation set** is a set of relationships:
  - a **relationship** is a connection between entities
    - ie. edges between entities
  - relationship set is a set of relations of the same kind
    - denoted by diamonds in ER
  - relationships can also have attributes

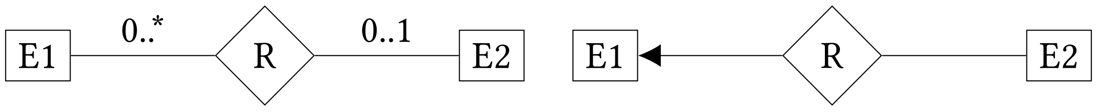


Figure 1: General Cardinality Notation

- the **cardinality** of a relationship is how many times entities participate in a relationship:
  - in a **one-to-one** relationship:
    - entities on either side can participate in at *most* 1 relationship
    - eg. dorm room bed assignments
    - denoted by arrows on both ends of the relationship diamond
  - in a **one-to-many** relationship:
    - entites on one side can participate in more than one relationship
    - eg. faculty and classes, person and pets
    - denoted by an arrow on the “one” side ie. the side participating in multiple relationships
  - in a **many-to-many** relationship:
    - entites on both sides can participate in more than one relationship
    - eg. students and classes
    - denoted by no arrows on either side
  - in **total participation**, every entity participates in the relationship *at least* once
    - denoted by a double-line on the side of the relationship diamond
  - for general cardinality notation, label an edge with $a..b$:
    - $*$ indicates unlimited cardinality
    - indicates the entity participates in the relationship between a through b times, inclusive
    - eg. in Figure 1, the two ER models are equivalent

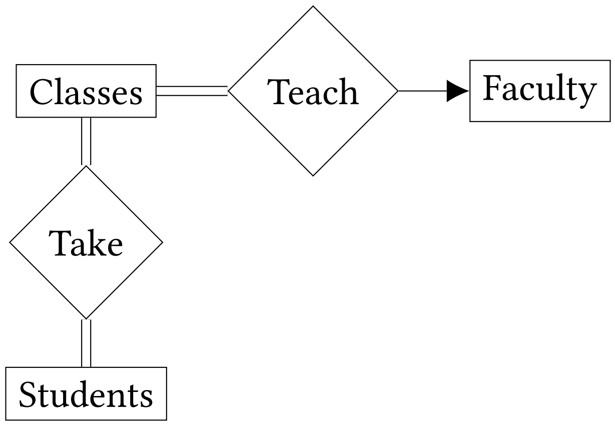
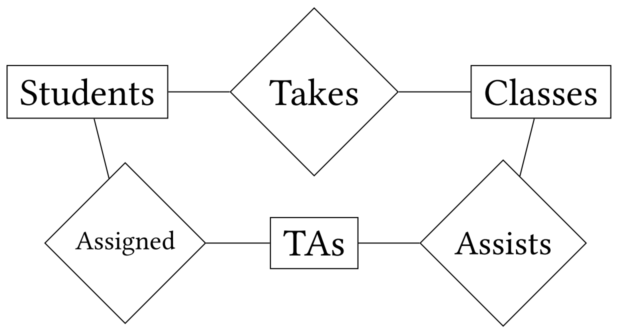


Figure 2: Relationship Notation Example

- ex. What do the relationships in Figure 2 mean?
  - a faculty member can teach multiple classes

- every class must be taught by a faculty member
- every student must take at least one class and every class must be taken by at least one student



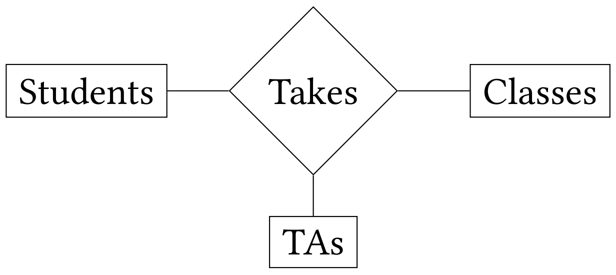


Figure 3: Tertiary Relationship Example

- sometimes, it is necessary to have more than just a binary relationship:
  - need an N-ary relationship where relationships connect N entites instead of just 2
  - ex. As seen in Figure 3:
    - if every TA is only assigned to assist a class, we can use purely binary relationships
    - however, if each student is *also* assigned to a particular TA, we may first try to add another binary relationship between student and TAs
      - but this leads to *redundancy* since the TAs are related to students through *both* their assignment to the class and student
    - instead, use a triple ie. tertiary relationship that encapsulates all of the information
      - the `Takes` relationship would hold `(sid, TA, class)`

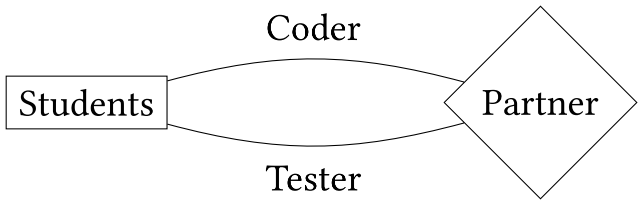


Figure 4: Role Example

- we can designate a **role** to each entity set that participates in a relationship set:
  - useful if an entity set participates more than once in the same relationship
  - denoted by labels on edges of a relationship in ER eg. in Figure 4
- may have superclasses and subclasses in ER:
  - ie. specializations and generalizations

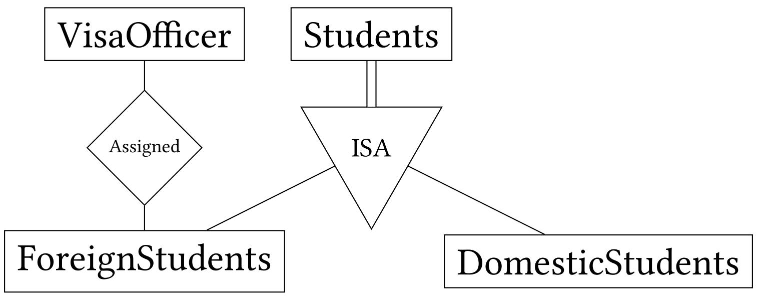


Figure 5: Subclass Example

- subclass inherits all attributes of its superclass
- subclass participates in the relationships of its superclass
- subclass may participate in its own relationship
- use an "IS-A" relationship in ER that connects superclass and subclass as seen in Figure 5:
  - denoted by a triangle that spreads into subclasses
  - by convention, top class is the superclass
- in **total specialization**, entity is *always* one of its subclasses:
  - denoted with double lines in ER
  - eg. *every* student is either a foreign or domestic student
- note that a superclass may belong to multiple subclasses
  - to make subclasses distinct, we can add a "disjoint" keyword under the triangle

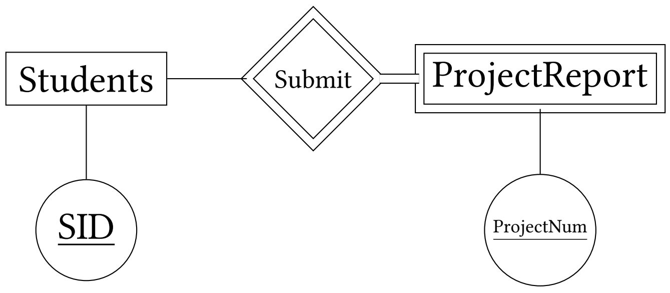


Figure 6: Weak Entity Set Example

- a **weak entity set** is one without a unique key:
  - denoted by a double rectangle and double diamond
  - *part* of its key will come from one or more entity sets it is linked to:
    - the **owner entity set** is the entity set providing part of the key
    - the **identifying relationship** is the relationship between a weak and owner entity set
    - needs a double edge between weak entity set and identify relationship because total participation is required to for the key to non-null
    - the **discriminators** are the attributes in a weak entity set that will also become part of the key

    · denoted by dotted underlines
  - compared to the typical **strong entity set**
  - ex. As seen in Figure 6, a project report is submitted by a student but only has a project number attribute
    * owner entity set is the student, identifying relationship is the project submission, and the discriminator is the project number

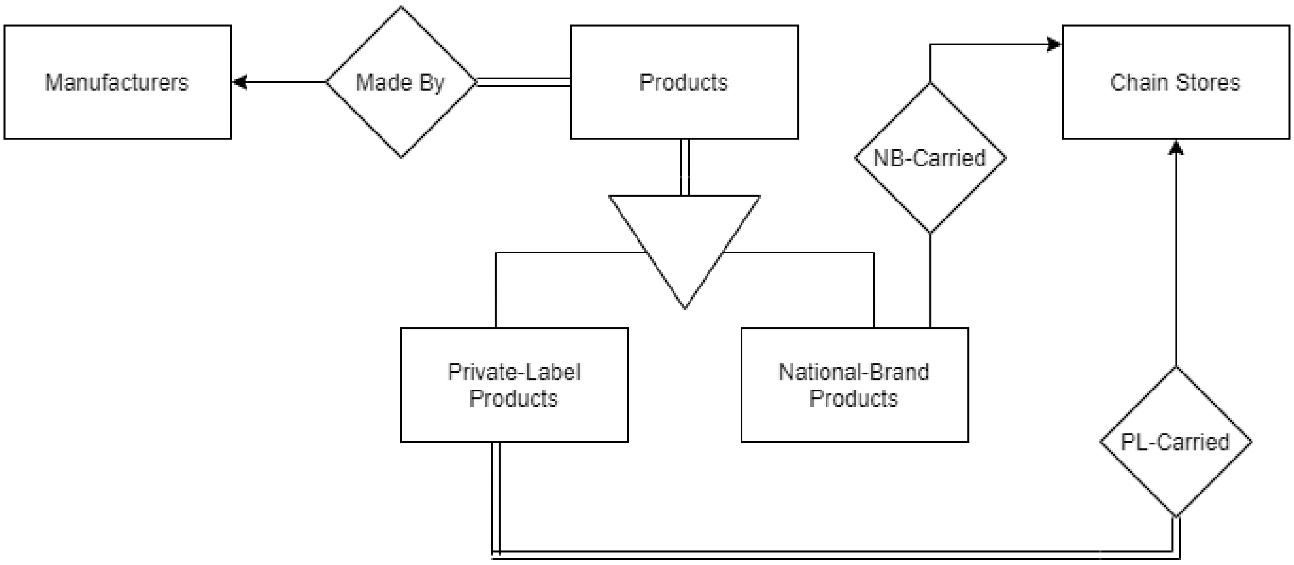


Figure 7: Barebones ER Example without Attributes

- ex. Encapsulating the following database design for stores and products in the ER as seen in Figure 7:
  - All products are either private-label or national-brand.
  - Every product is manufactured by exactly one manufacturer.
  - Every private-label product is carried by exactly one chain store.
  - Some national-brand products may not be carried by any chain store.
- ex. Creating a new entity vs. simply encapsulating it as attributes:
  - if `Faculty(name, addr)` are instructors of `Class(dept, cnum, title)`, do we need a separate entity for faculty?
  - if it is a one-to-one relationship, it is usually better to encapsulate the entity as an attribute
  - for many-to-many or many-to-one relationships, it is better to separate out the entity to prevent redundancy in attributes

ER Model Conversion

- converting an ER model to tables is mostly straightforward:
  - automatic conversion tools exist
  1. for every strong entity set, convert it into a table with all the attributes

2. for each relationship set, create one table with keys from the linked entity sets and its own attributes:
   - name conflicts should be prefixed with the entity name
   - if there are roles, those roles are preferred over the conneted entity's key
3. for every weak entity set, create one table with its own attributes and keys from the owner entity set
   - there is *no* table for identifying relationship set, but its attributes are also added to the table
4. for every subclass relationship, there are two approaches:
   1. create one table for each subclass with its own attributes and the key of its superclass
      - requires a join to connect the information in the subclass with its superclass
   2. alternatively, create one gigantic table for the super class that includes all atrributes
      - requires using null values for missing attributes
- note that this initial conversion loses the cardinality information from the ER model

Converting Figure 8 to relations:

```
-- entity sets
Student(PRIMARY KEY name, addr)
TA(PRIMARY KEY name, addr)
Class(PRIMARY KEY cnum, title)
Faculty(PRIMARY KEY name, addr)

-- relationship sets
Teach(PRIMARY KEY cnum, name)
Take(PRIMARY KEY quarter, PRIMARY KEY Student_name,
     PRIMARY KEY TA_name, PRIMARY KEY cnum)
Partner(PRIMARY KEY coder, tester)
ProjectReport(PRIMARY KEY num, grade, PRIMARY KEY name)

-- subclasses
ForeignStudent(PRIMARY KEY name, country)
HonorStudent(PRIMARY KEY name, fellowship)
-- alternatively, Student(PRIMARY KEY name, addr, country, fellowship)
```

- to find the keys in a relationship, consider the edges in the relationship graph:
  - each edge represents a relationship
  - eg. in many-to-one graph between classes and faculty, the class key will uniquely identify the relationship

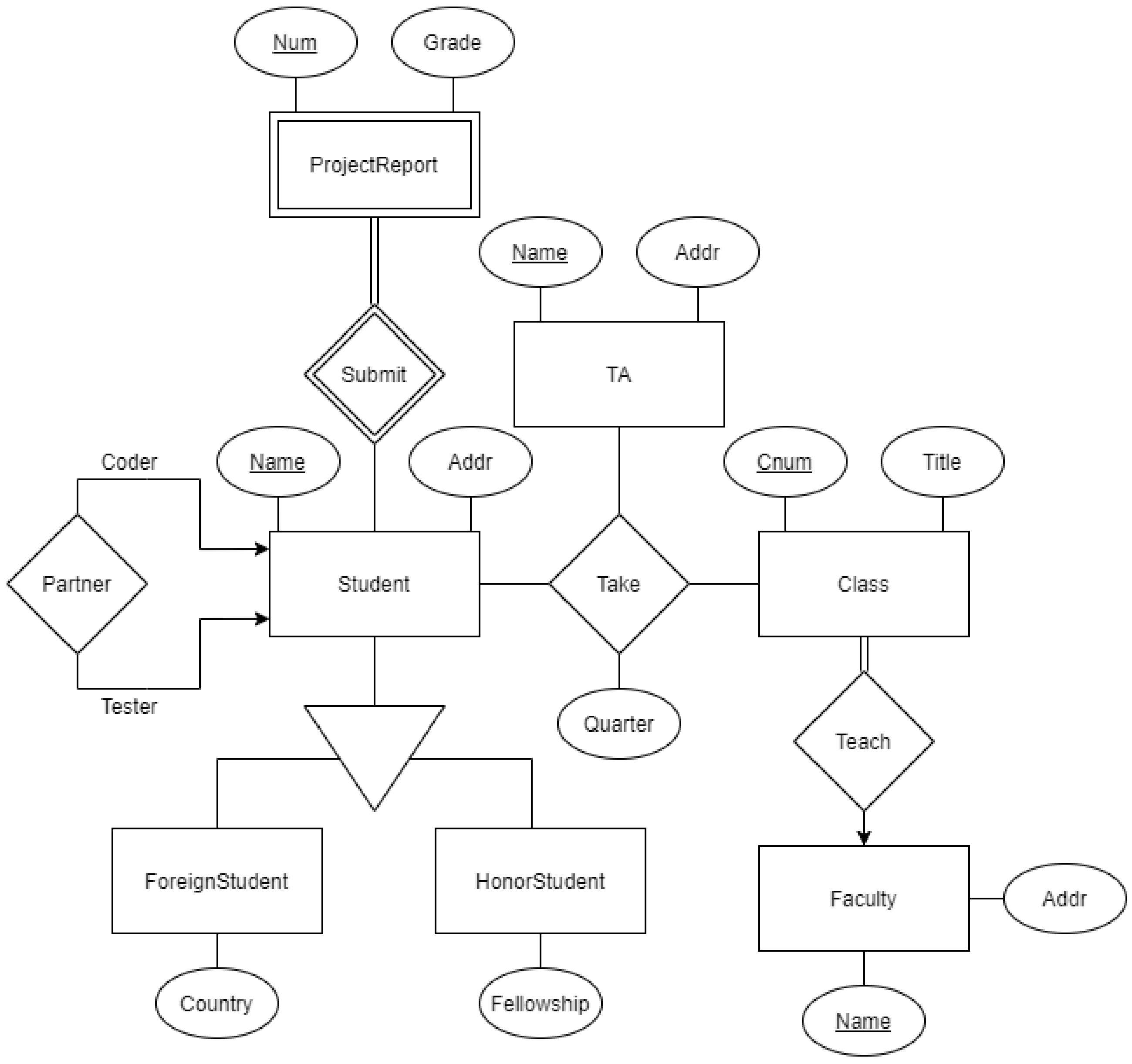


Figure 8: ER Conversion Example

    - eg. in one-to-one graph between students, either students key can uniquely identify the relationship

Normalization Theory

- how do we design *"good"* tables for a relational database?
  - typically, we start with ER and convert it into tables
  - however, there are many choices to make in ER that lead to different tables
  - **relational design theory** aka **normalization theory** is a theory on what are "good" table designs:
    - hinges on trying to minimize *redundancy* in table design
    - provides an algorithm that converts "bad" designs into "good" designs automatically
  - note that performance may be another metric that conflicts with minimizing redundancy
- redundancy leads to potential anomalies:
  1. when updating, information may be updated partially and inconsistently
     - eg. a student changes their address
  2. when inserting, we may not include some information at all
     - eg. a student does not take any class
  3. when deleting, we may delete other information
     - eg. the *only* class a student takes gets cancelled
- ex. Consider the following table design:
  - `StudentClass(sid, name, addr, dept, cnum, title, unit)`
  - should this be split into multiple entities?
    - yes, we can see that information corresponding to students and classes will both be redundantly represented in the design
    - but could lead to faster information lookup if we want to associate classes to students
  - tradeoff between redundant storage vs. access efficiency
  - could split into the following tables
    - `Student(sid, name, addr)` , `Class(dept, cnum, title, unit)` , and `Enroll(sid, dept, cnum)`
- is there an algorithm to arrive at the better design more systematically?
  - need to determine where the redundancy stems from ie. which attributes in which tuples can be "hidden" without losing information:
    - some attributes are uniquely *determined* by other attributes
    - eg. $sid \to (name, addr)$ and $(dept, cnum) \to (title, unit)$
      - note that the way we split the tables corresponds to these de-

terminations, in addition to adding a new table that connects the two determinations
    * uses the concept of functional dependencies in normalization theory
    * note that these dependencies and other relevant properties would be communicated to database designers via the domain experts who understand the data clearly
  - when there is a functional dependency due to this determination, we may have redundancy
    * can decompose these tables into smaller ones to reduce redundancy, no need to store multiple instances of the relationship

Functional Dependencies

- for a **functional dependency (FD)** $X \rightarrow Y$:
  - $\forall\ u_1, u_2 \in R$, if $u_1[X] = u_2[X]$ then $u_1[Y] = u_2[Y]$
    * ie. no two tuples in R can have the same X values but different Y values
  - the notation $u[X]$ gives the values for the attributes X of tuple u
    * eg. $u[sid, name] = (100, James)$
  - ex. For `StudentClass` :
    * $sid \rightarrow name$ is a functional dependency
    * $dept, cnum \rightarrow title, unit$ is a functional dependency
    * $dept, cnum \rightarrow sid$ is *not* a functional dependency
  - ex. For Table 8 and Table 9, $AB \rightarrow C$ is a valid functional dependency
    * however, Table 10 does not have the same functional dependency
- types of FD:
  - an FD $X \rightarrow Y$ is a **trivial FD** when $Y \subseteq X$
    * *always* true regardless of real world semantics
  - an FD $X \rightarrow Y$ is a **nontrivial FD** when $Y \not\subseteq X$
  - an FD $X \rightarrow Y$ is a **completely nontrivial FD** when $X \cap Y = \emptyset$

Table 8: Example Relation

A	B	C
a_1	b_1	c_1
a_1	b_2	c_2
a_2	b_1	c_3

Table 9: Example Relation

A	B	C
a_1	b_1	c_1
a_1	b_2	c_2
a_2	b_1	c_1

Table 10: Example Relation

A	B	C
a_1	b_1	c_1
a_1	b_1	c_2
a_2	b_1	c_3

- through **logical implication**, a *set* of FDs may imply a new FD:
  - ex. With table $R(A, B, C, G, H, I)$ and a set of FDs $\{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$
    * $A \rightarrow H$ is true since $A \rightarrow B \rightarrow H$ ie. we say A logically implies H
- the **canonical database** is a method to check logical implication:
  - to check logical implication $A \rightarrow H$:
    * assume a random tuple u_1 with random values for each attribute
    * assume a second random tuple u_2 with random values except $u_2[A] = u_1[A]$
    * want to show $u_2[H] = u_2[H]$
    * $u_1[B] = u_2[B]$ from $A \rightarrow B$
    * $u_1[C] = u_2[C]$ from $A \rightarrow C$
    * $u_1[H] = u_2[H]$ from $B \rightarrow H$
    * given assumption of matching A, these are the only four values that are logically implied
  - to check logical implication $AG \rightarrow I$:
    * assume a random tuple u_1 with random values for each attribute
    * assume a second random tuple u_2 with random values except $u_2[A] = u_1[A]$ and $u_2[G] = u_2[G]$
    * want to show $u_2[I] = u_2[I]$
    * $u_1[B] = u_2[B]$ from $A \rightarrow B$
    * $u_1[C] = u_2[C]$ from $A \rightarrow C$
    * $u_1[H] = u_2[H]$ from $B \rightarrow H$
    * $u_1[I] = u_2[I]$ from $CG \rightarrow I$
- thus the **closure** of a functional dependency set F is $F+$:
  - represents the set of all FDs that are logically implied by F

- while the closure of the attribute set X is $X+$
  - represents the set of all attributes that are functionally determined by X
- closure $X+$ computation algorithm:
  1. start with $X+ = X$
  2. repeat until no change in $X+$:
     - if there is $Y \to Z$ and $Y \subseteq X+$, then $X \leftarrow X+ \cup Z$
- ex. What is the attribute set closure $\{sid, dept, cnum\}+$ for $StudentClass$ given the FDs $\{sid \to name, (dept, cnum) \to (title, unit)\}$?
  - $\{sid, dept, cnum, name, title, unit\}$
- ex. With table $R(A, B, C, G, H, I)$ and a set of FDs $F = \{A \to B, A \to C, CG \to H, CG \to I, B \to H\}$:
  - $\{A\}+ = \{A, B, C, H\}$
  - $\{A, G\}+ = \{A, G, B, C, H, I\}$
    - $\{A, G\}$ is a *key* of R, while $\{A\}$ or $\{A, B\}$ are not
- thus can create a new formal definition for a **key** X of R:
  1. $X+ = R$
  2. no subset of X satisfies (1) ie. X is minimal

- projecting functional dependencies:
  - ex. For $R(A, B, C, D)$ and a set of FDs $F = \{A \to B, B \to A, A \to C\}$, what FDs hold for $R'(B, C, D)$, where R' is a projection of R?
    - $\{B \to C\}$, need to consider the closure $F+$ and pick the FDs that hold on the projected table

Decomposition

- now, we can remove redundancy by perform a series of decompositions on the ER model using FDs
  - in general, want to split $R(A_1, \dots, A_n) \to R_1(A_1, \dots, A_i), R_2(A_j, \dots, A_n)$ such that $\{A_1, \dots, A_n\} = \{A_1, \dots, A_i\} \cup \{A_j, \dots A_n\}$
  - need to be careful not to *lose* information when performing the decompositions:
    - ie. perform *lossless* decomposition
    - a decomposition of R into R_1, R_2 is a **lossless-join decomposition** if and only if $R = R_1 \bowtie R_2$
      - ie. can always get back the original table R if needed
    - but we need a more concrete general condition to guarantee the lossless-join decomposition in implementation

Table 11: Student Table

cnum	sid	name
143	1	James
143	2	Elaine
324	3	Susan

Table 12: S_1

cnum	sid
143	1
143	2
325	3

Table 13: S_2

cnum	name
143	James
143	Elaine
325	Susan

Table 14: $S_1 \bowtie S_2$

cnum	sid	name
143	1	James
143	1	Elaine
143	2	James
143	2	Elaine
325	3	Susan

- ex. In Table 11, is the decomposition into $S_1(cnum, sid), S_2(cnum, name)$ lossless?
  - no, $S_1 \bowtie S_2$ does not equal the original table, as seen in Table 14
    - number of tuples actually increased
- ex. In Table 11, is the decomposition into $R_1(cnum, sid), R_2(sid, name)$ lossless?
  - yes, $R_1 \bowtie R_2$ would equal the original table
  - the lossless quality is met because each tuple only joins with *exactly one* tuple of the other table:

    * ie. each decomposing table to needs have a unique identifying attribute
    * unlike in Table 14
- thus we can formalize that decomposition $R(X, Y, Z) \to R_1(X, Y), R_2(Y, Z)$ is lossless-join if $Y \to X$ or $Y \to Z$:
  - ie. a shared attribute uniquely determines every remaining attribute in *one* of the remaining tables
  - ie. the shared attribute is the *key* of one of the decomposed tables
    * only needs to be the key of one for the join condition to be met
  - note that a single table may be decomposed into many tables, but the decomposition is typically performed iteratively, into two tables at a time
- ex. Decomposing `StudentClass` into `Student(sid, name, addr)` and `Class(sid, dept, cnum, title, unit)` :
  - with the FD $sid \to (name, addr)$, the shared attribute is the key of one of the resultant tables
  - thus the decomposition is lossless-join
- functional dependencies have potential to cause redundancy ie. repeated information in tuples:
  - eg. `StudentClass(sid, name, addr, dept, cnum, title, unit)` has redundancy caused by the FD $sid \to (name, addr)$
  - however, sometimes FDs does not lead to redundancy
    * eg. in `Student(sid, name, addr)` , the same FD $sid \to (name, addr)$ does *not* lead to redundancy
  - whenever the LHS of a FD contains the key, the FD does not cause redundancy
    * this is because the RHS will only be repeated once since the tuples referred to by the LHS are unique
- a relation R is in **Boyce-Codd normal form (BCNF)** with regard to the set of FDs $F+$ if and only if for every nontrivial functional dependency $(X \to Y) \in F$, X is a key of R:
  - informally, this normal form indicates a "good" table design
  - ensures there is no redundancy in the table due to FD
  - there are other definitions for normal forms eg. third and fourth normal forms
    * but BCNF AKA 3.5th normal form is the most useful and widely used
- ex. Are the following examples in BCNF?
  - `Class(dept, cnum, title, unit)` with FD $(dept, cnum) \to (title, unit)$:
    * check closure of the LHS of the FD
    * does not violate BCNF since LHS is a key after calculating closure
  - `Employee(name, dept, manager)` with F set $\{name \to dept, dept \to$

$manager\}$:
    - can translate the FDs as each employee is in only one department and each department only has one manager
    - intuitively, looks like a *reasonable* table design?
    - check BCNF by checking closures:
        - $\{name\}+ = \{name, dept, manager\}$ is a key
        - $\{dept\}+ = \{dept, manager\}$ is *not* a key
    - not in BCNF, has dependencies!
        - manager is the same for everyone in the department
        - would be better to decompose into two tables
- whenever there is a BCNF violation, perform *decomposition* until all tables are in BCNF using the following algorithm:
    - for all R in the schema, identify all non-trivial dependencies $X \rightarrow Y$ that hold on R and X is not a key:
        1. compute $X+$
        2. decompose R into $R_1(X+)$ and $R_2(X, Z)$, where X are common attributes and Z are all attributes in R except $X+$
            - this decomposition guarantees a lossless join since the shared attribute is the key of one of the decomposed tables
    - repeat until no more decomposition
    - note that this BCNF decomposition algorithm does not always lead to a unique set of relations
        - depends on ordering
- ex. Decompose the following example:
    - `ClassInstructor(dept, cnum, title, unit, inst, office, fax)` with FD set:

$$F = \begin{cases} inst \rightarrow office \\ office \rightarrow fax \\ (dept, cnum) \rightarrow (title, unit) \\ (dept, cnum) \rightarrow inst \end{cases}$$

    1. check `ClassInstructor` in BCNF with $\{inst\}+ = \{inst, office, fax\}$
        - not in BCNF, split `ClassInstructor` into `R1(inst, office, fax)` and `R2(dept, cnum, title, unit, inst)`
    2. check R_1 in BCNF with $\{office\}+ = \{office, fax\}$
        - not in BCNF, split R_1 into `R3(office, fax)` and `R4(inst, office)`
    3. check R_2 in BCNF with $\{dept, cnum\}+ = \{dept, cnum, title, unit, inst\}$:
        - note that we do not need to check the FD $inst \rightarrow office$ or $office \rightarrow fax$ since those do not hold in R_2
        - in BCNF, LHS is indeed a key
    - decomposing into three tables is now BCNF

        * intuitively, R_3 represents fax machines in an office, while R_4 represents the instructor in an office and R_2 represents course details
- normalization splits tables to reduce redundancy:
    - however, splitting tables has negative performance implications
    - as a rule of thumb, start with normalized tables and merge them if performance is not good enough
        * examine the particular queries that are costing performance and join back those tables accordingly

Database Integrity

- how can we ensure that data in the database in consistent?
    - many attributes could be invalid in a table
        * eg. could have negative GPAs, invalid addresses, ages, course numbers, SIDs, or units, etc.
- multiple places to check data integrity
    - eg. DBMS, application server, user-facing client
- in the RDBMS, there are three ways to enforce data validity:
    1. domains eg. GPA is real
    2. constraints that generate an error and abort the offending SQL statement
        - eg. key constraints, referential integrity, `CHECK` constraints
    3. SQL trigger from SQL99
        - acts as an **event-condition-action (ECA)** where when a certain event happens, invoke an action to handle it
- for key constraints as previously detailed:
    - specifies a set of attributes that should be unique in a table
    - one primary key can be specified per table, but multiple unique keys can be specified

Referential Integrity

- **referential integrity** involves particular values in one table *referencing* ones in another table:
    - eg. if SID appears in the `Enroll` table, it should also appear in the `Student` table:
        * similarly for classes in `Enroll` and `Class`
        * but note that the reverse conditions are *not* true
    - thus the referential integrity constraint is *directional*

  - terminology where if `E.B` references `S.B` :
    * `E.B` is the **foreign key** ie. referencing attribute
    * `S.B` is the **referenced attribute**
  - conditions for foreign keys:
    * the refernced attribute should always exist in the referenced table
    * when foreign key is `NULL` , no referential integrity check is performed
    * referenced attributes must be primary key or unique
  - instantiation restrictions:
    * since the table definition references other tables, these referenced tables should be made first
      · alternatively, could make the table without constraints and add the constraints later
    * in addition, need to insert tuples into the referenced tables *first* when inserting into the referencing table

Enrollment table with foreign keys:

```
CREATE TABLE Enroll(
    sid  INT,
    dept CHAR(2),
    cnum INT,
    sec  INT,
    FOREIGN KEY(sid) REFERENCES Student(sid),
    -- sid INT REFERENCES Student is alternative shorthand,
    -- optional to repeat matching attributes
    FOREIGN KEY(dept, cnum, sec) REFERENCES Class
);
```

- when can RI be violated when table `E` references a key of `S` ?
  - the modifying operations on a table can be an insertion, deletion, or update
  1. insertion into `E`
  2. update on `E`
  3. deletion from `S`
  4. update on `S`
- two modification operations can violate the referential integrity:
  - the DBMS always rejects any violating constraint by default
  - however, if the violation comes from the referenced sides, we can instruct DBMS to automatically fix the violation:
    * how should DBMS fix a deletion from `S` ?
      · `ON DELETE CASCADE` deletes referencing tuples from `E` since it becomes a dangling tuple
      · `ON DELETE SET NULL` sets those tuples to have a null reference

        * how should DBMS fix an update from `S` ?
            - `ON UPDATE CASCADE` updates the referencing tuples
            - `ON UPDATE SET NULL` sets those tuples to have a null reference
        * `SET DEFAULT` can also be used to set a default value on referenced key update or deletion
        * note that these "fixes" are applied to the table `E` , but are really monitoring changes to `S`
    - RI is the only SQL constraint that can fix itself:
        * other constraints simply reject the statement and generate an error
        * note that not all DBMS support all the "fixing" actions
    - note that if referenced attributes were *not* unique, adds complications to `ON DELETE` and `ON UPDATE`

Allowing the DBMS to fix referential integrity violations:

```
CREATE TABLE E(
    A INT, B INT,
    FOREIGN KEY(B) REFERENCES S(B),
    ON UPDATE { CASCADE | SET NULL | SET DEFAULT },
    ON DELETE { CASCADE | SET NULL | SET DEFAULT }
);
```

Table 15: A Self-Referencing Table

A	B
1	`NULL`
2	1
3	2
4	3

- complications with relational integrity:
    - Table 15 is a self-referencing table `T` with `FOREIGN KEY(B) REFERENCES T(A)` :
        * if we have the `ON DELETE CASCADE` rule, what happens when `(1, NULL)` is deleted?
        * the entire database will be cleared
    - in a circular constraint:
        * `ChickenFrom(cid, eid)` and `EggFrom(eid, cid)`
        * `Chicken.eid` is foreign key for some `Egg.eid` and `Egg.cid` is foreign key for some `Chicken.cid`
        * to create the tables:
            - need to create the table without the foreign key constraint and then alter the table

      * inserting tuples:
        1. create the first chicken or egg that came from nowhere ie. the foreign key is null
        2. create a chicken and egg that came from itself

Instantiating a circular constraint and inserting tuples:

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

-- creating first chicken or egg from nowhere
INSERT Into Chicken VALUES (1, NULL);
INSERT Into Egg VALUES (1, NULL);

-- chicken and egg from itself
INSERT INTO Chicken VALUES (1, NULL);
INSERT INTO Egg VALUES (1, 1);
UPDATE Chicken SET eid=1 WHERE eid IS NULL;
```

`CHECK` Constraint

- the `CHECK(condition)` clause allows a certain condition to be checked:
  - condition can be complex and may include subqueries
  - constraint is attached to a *particular* table
    * whenever the table is updated, the statement is rejected if the condition is violated

1. Enforce the GPA to be between 0.0 and 4.0:

```
CREATE TABLE Student(
    sid INT, name VARCHAR(50), addr VARCHAR(50), GPA REAL,
    CHECK(GPA≥0 AND GPA≤4)
);
```

2. Enforce class number to be less than 600 and class units to be below 10:

```
CREATE TABLE Class(
    dept CHAR(2), cnum INT, sec INT, unit INT,
    title VARCHAR(100), instructor VARCHAR(100),
    CHECK(cnum<600 AND unit≤10)
);
```

3. Enforce the units of all CS classes to be greater than 3:

```
CREATE TABLE Class(
    dept CHAR(2), cnum INT, sec INT, unit INT,
    title VARCHAR(100), instructor VARCHAR(100),
    CHECK(dept<>'CS' OR unit>3) -- can write logical implication A→B as ~A+B
);
```

4. Enforce students whose GPA is less than 2 to not take CS classes:

```
CREATE TABLE Enroll(
    sid INT, dept CHAR(2), cnum INT, sec INT,
  CHECK(dept<>'CS' OR EXISTS (SELECT * FROM Student S WHERE sid=S.sid AND GPA≥2))
    -- alternatively,
    -- CHECK(dept<>'CS' OR sid NOT IN (SELECT sid FROM Student WHERE GPA<2))
);
```

- but the `CHECK` constraint doesn't check when the student's GPA changes *after* enrolling
    - is acceptable in this case, but is situational

5. Express an RI constraint using the `CHECK` constraint:

```
CREATE TABLE Student(...);
CREATE TABLE Enroll(
    sid INT, dept CHAR(2), cnum INT, sec INT,
    CHECK(sid IN (SELECT sid FROM Student))
);
```

- but the `CHECK` constraint will not catch updates on the student table
    - can *simulate* a referential integrity check, but does not perform exactly the same

Trigger

- a **trigger** is an **event-condition-action (ECA)** rule:
    - part of SQL3 standard
    - allows explicitly specifying what events to monitor, what condition to check, and what action to take if the condition is met

General trigger syntax:

```
CREATE TRIGGER <name>
{BEFORE | AFTER} {INSERT | DELETE | UPDATE [OF A1...An]} ON R
    [REFERENCING {OLD | NEW} {TABLE | ROW} AS <var>
```

```
     FOR EACH {ROW | STATEMENT}]
[WHEN (<condition>)]
[BEGIN]
    <actions>
[END]
```

1. If a student's GPA goes below 2, drop the student from all classes:

```
CREATE TRIGGER MinGPA
AFTER UPDATE OF GPA ON STUDENT -- event
    REFERENCING NEW ROW AS S   -- reference the triggering row
    FOR EACH ROW               -- row-level trigger
WHEN (GPA < 2)                 -- condition
BEGIN                          -- action
    DELETE FROM Enroll WHERE sid=S.sid
END; -- BEGIN / END is optional with a single action statement
```

- the `REFERENCING` clause allows us to reference some state related to the triggering event:
  - `NEW ROW` for triggering row, or `OLD ROW` for the previous row
  - `FOR EACH ROW` makes the trigger a **row-level trigger** that runs for each row
  - `FOR EACH STATEMENT` makes the trigger a **statement-level trigger** that runs for each row
    * use `OLD TABLE` or `NEW TABLE` to reference multiple updates rows
  - note that when deleting, we only have access to the previous row or table

2. All new students have to take CS143:

```
-- row-level trigger:
CREATE TRIGGER ForceCS143
AFTER INSERT ON Student
    REFERENCING NEW ROW AS NEW_S
    FOR EACH ROW
INSERT INTO Enroll VALUES (NEW_S.sid, 'CS', 143, 1);

-- statement-level trigger:
...
    REFERENCING NEW TABLE AS NEW_S
    FOR EACH STATEMENT
INSERT INTO Enroll (SELECT sid, 'CS', 143, 1 FROM NEW_S);
```

3. Recursive triggers:

```sql
CREATE TRIGGER Recursion
AFTTER INSERT ON R
INSERT INTO R VALUES(1);
-- infinite recursion!
```

MySQL Support

- the key constraint is supported
- under InnoDB, most referential integrity constraints are supported:
  - does not support `SET DEFAULT` on updates or deletes
  - does not support single-column `REFERENCES` shorthand
    - cannot omit column names even if the names are the same
- no `CHECK` constraint support in standard MySQL
  - MariaDB 10.2.1 supports a limited `CHECK` constraint
- limited support for triggers
  - does not allow updating the table that caused the trigger event
- note that MySQL will silently ignore constraints it does not support:
  - no warning or error messages
    - need to check constraints are being properly applied
  - safer to use the most conservative syntax

Non-Relational Databases

MongoDB

- **JavaScript object notation (JSON)** started as syntax to represent objects in JS:
  - eg. `[{"x": 3, "y": "Good"}, {"x": 4, "y": "Bad"}]`
  - popularity of JSON has grown alongside JS:
    - simple and easy to learn
    - JSON data is mostly self-descriptive
  - most popular data-exchange formats are JSON, XML, CSV
- supports basic data types like number and strings, as well as arrays and objects:
  - eg. `"best", "worst"`
  - eg. `[1, 2, 3, "four", 5]`
  - objects are just attribute and name pairs denoted with curly braces:
    - eg. `{"sid": 301, "name": "James Dean"}`
    - objects can be nested
- the JSON model is based on a tree ie. hierarchical data model:
  - cannot use relational databases unless we map the tree into a relation
  - *pros*:
    - perfect for representing objects
  - *cons*:
    - too complex for representing analytical or transactional data
- how to use RDBMS with JSON?
  1. simply store object's JSON as a string in a column:
     - usually have at least a key to index into the corresponding JSON
     - this allows for simple lookups with the JSON, but not for more complicated queries
  2. normalize the object into the set of relations
     - ie. converting the JSON into relational tables
  - can we instead create a "native" database for JSON?
- **MongoDB** is a database for JSON objects:
  - AKA a **NoSQL** database
  - the data is stored as a collection of documents:
    - a **document** is essentially a JSON object, like rows in RDB
    - a **collection** is a group of similar documents, like tables in RDB

Example MongoDB document:

```
{
    "_id": ObjectId(123456abc),
    "title": "MongoDB",
    "description": "MongoDB is NoSQL database",
    "tags": ["mongodb", "database"],
    "likes": 100,
    "comments": [
        { "user": "lover", "comment": "perfect" },
        { "user": "hater", "comment": "worst" }
    ]
}
```

- the `_id` field is a primary key:
  - may be of any type other than array
  - if not provided, automatically added by MongoDB with a unique `ObjectId` value
- in RDB, every column is atomic:
  - in MongoDB, each field can be an array or nested object
  - can also be stored as **binary representation of JSON (BSON)**:
    - supports more data types than JSON
    - does not require double quotes for field names
- MongoDB philosophy:
  - adopts JavaScript "laissez-faire" philosophy:
    - not too strict, but instead accommodating
    - handle user requests in a *reasonable* way
  - *schema-less* with no predefined schema
    - one collection will store documents of any kind with no complaint
  - no need to "plan ahead":
    - a database is created when a first collection is created
    - a collection is created when a first document is inserted
  - this leniency has both advantages and disadvantages
- MongoDB CLI API:
  - `use database`
    - select a specific database
  - `db.collection.insertMany(arr)` :
    - insert the documents specified in `arr` into `collection` in `db`
    - also `insertOne`
  - `db.collection.find(cond)` :
    - analogous to SQL select, finds all documents matching the condition
    - eg. `db.books.find({likes: 100})`
    - eg. `db.books.find({likes: {$gt: 1000}})`

        - other boolean operators include `$and, $or, $not, $lt, $ne`
      - also `findOne`
  - `db.collection.updateOne(cond, operation)` :
      - update the matching conditions by performing an operation
      - eg. `db.books.updateOne({title: "MongoDB"}, {$set: {likes: 200}})`
        - other update operations include `$inc, $unset`
      - also `updateMany`
  - `db.collection.deleteOne(cond)` :
      - delete matching documents
      - eg. `db.books.deleteOne({title: "a"})`
      - also `deleteMany`
  - `db.collection.drop()`
      - database will disappear if all of its collections are removed
- MongoDB aggregates:
  - very similar to general SQL selection queries, not just SQL aggregates
  - made up of various **stages** similar to each individual clause of a SQL statement
      - the entire aggregate **pipeline** is similar to the overall SQL statement
  - common aggregate stages:
      - `$match` performs like SQL `WHERE`
      - `$group` performs like SQL `GROUP BY`
      - `$sort` performs like SQL `ORDER BY`
      - `$limit` performs like SQL `FETCH FIRST`
      - `$project` performs like SQL `SELECT`
      - `$unwind` replicates the document per every element in the array specified by the field path
        - eg. `{$unwind: "$y"}` would transform the document `{x: 1, y: [1,2]}` into `{x: 1, y: 1}` and `{x: 1, y: 2}`
      - `$lookup` performs a "lookup and join" with another document based on the attribute value
        - eg. `{$lookup: {from, localField, foreignField, as}}`
  - note that aggregate stages can be done in different orders, and can be repeated

Example aggregate:

```
db.orders.agggregate([
    { $match: { status: "A" }},      // WHERE
    { $group: {
        _id: "$cust_id",             // GROUP BY
        total: { $sum: "$amount" }, // aggregate functions
        count: { $sum: 1 }
    }},
    { $sort: { total: -1}}          // ORDER BY
]);
```

- comparing with RDB:
  - *pros*:
    - preserves the structure of nested objects
  - *cons*:
    - has potential redundancies unlike in relational models
    - restructuring or combining data is complex and inefficient
      - unlike relational operators that could be easily combined in relational models
  - MongoDB is much less strict than RDB

Map Reduce

- motivation of distributed analytics for non-relational DB using clusters:
  - often, our data is *non-relational* and huge
    - eg. billions of query logs or web pages
  - want to be able to run multiple queries on *distributed* machines for better performance
    - fast big data analytics
  - transform and load data into RDBMS?
    - data transformation is data-specific and labor intensive
    - going from non-relational DB to distributed RDBMS using data transformation is cost prohibitive
- ex. Perform search log analysis:
  - given logs of billions of queries, count the frequency of each query:
    - eg. useful for search algorithms
    - log file is spread over many machines, records keyword query and frequency of searches
  - operation is simple, but how can we run it on thousands of machines in parallel?
    - need to process each query log *independently*, and then *combine* results
    - first, on each machine, transform the large log file into a certain output
      - eg. map log file into queries and their respective frequencies on each individual machine
    - then, sum up the counts of different machines together
- generalizing this process:

1. map / transform step:
   - eg. transform query log into `(query, freq)`
     * since each log records the times a query was searched, want to extract only the relevant data
   - importantly, this transformation can be done on multiple machines in *parallel*
2. reduce / aggregate step:
   - eg. group by query and sum up frequencies
   - importantly, the grouping operation can move tuples with the same key to the same machine:
     * ie. *reshuffling* tuples of the same key to the same machine
     * then allows aggregation to be done in *parallel*

- the map and reduction steps are the same, except for the specific mapping and reducing *functions*:
  * can set up a distributed infrastructure that allows programmers to specify mapping and reducing functions
    · handles automatic partitioning, distribution, speed disparity, etc.
  * this is the **MapReduce** model:
    · map function m where m : *unit data* $\rightarrow (k1, v1), (k2, v2), \ldots$
    · reduce function r such that $(k, v1), (k, v2), \ldots \rightarrow (k, r(v1, v2, \ldots))$

- ex. Build an inverted index from web pages for finding matching documents:
  - original documents are the web pages and their IDs
  1. transform original pages into `(keyword, docid)`
     - break down documents into their constituent words
  2. group by key word and concatenate the indexes into a list
- Hadoop was the first open source implementation of the Google file system and MapReduce:
  - implemented in Java
  - user implements `Mapper.map(key, val, output, reporter)`
    * as well as `Reducer.reduce(key, val, output, reporter)`
- Spark is an open source cluster computing infrastructure:
  - modern alternative to Hadoop
  - supports map-reduce as well as SQL
  - input data is converted into a **resilient distributed dataset (RDD)**:
    * a collection of independent tuples
    * tuples are automatically distributed and shuffled by Spark
  - supports multiple programming languages Scala, Python, etc.
- key Spark API:
  - transformation on RDD tuples:
    * `map` converts one input into one output tuple
    * `flatMap` converts one input into multiple output tuples

        * `reduceByKey` specifies how two input values of the same key should be aggregated
        * `filter` filters out tuples based on a condition
    - actions on RDD:
        * `textFile` creates RDD from text (each line becomes an RDD tuple)
        * `collect` creates Python tuples from Spark RDD
        * `saveAsTextFile` saves the RDD in a directory as text files

Example Spark program to count words:

```
lines = sc.textFile('input.txt') # sc is Spark context object
words = lines.flatMap(lambda line: line.split(' '))
word1s = words.map(lambda word: (word, 1))
wordCounts = word1s.reduceByKey(lambda a, b: a+b)
wordCounts.saveAsTextFile('output')
```

Disk

- the **disk** is connected to the CPU through a disk controller and the system bus:
  - disk is permanent, non-volatile storage
    - may be a typical magnetic disk with platters or a solid state drive made up of semiconductors
  - separate structure from main memory:
    - need to load data from disk into the main memory so the CPU can act on it
      - typically done in a **block** granularity of size between 512B to 50KB, at several GB per second
    - on the other hand, transferring from main memory to CPU is done in a **word** granularity of size 1B to 64B, at tens of GB per second
- on a magnetic disk drive:
  - the entire disk is made up of a **cylinder** with multiple **platters**
  - platters are made up of concentric **tracks**
  - each track is made up of many **sectors**
    - sector is essentially the same as a block or page
  - data is transfered in a unit of blocks to amortize high access delay
- access time calculations:
  - what needs to be done to read a page?
    - need to move the disk arm to the correct track
    - arm needs to wait for the sector to spin under the head
  - thus, access time can be calculated as:

$$T_A = \textit{seek time} + \textit{rotational delay} + \textit{transfer time}$$

  - the average seek time is typically around 10 ms
  - disks typically rotate between 1000 to 15000 rpm
    - ex. with 6000 RPM, the average rotational delay is around $60\frac{1}{6000}\frac{1}{2} = 0.005$ seconds
  - transfer time is a function of the sector length and RPM
    - ex. with 10000 sectors per track at 6000 rpm the transfer time is $60\frac{1}{6000}\frac{1}{10{,}000} = 0.001$ ms
  - seek time dominates the calculation
    - if we are storing data on a magnetic disk, *must* ensure that when we read our data, it is done consecutively so that we do not have to adjust the head
- random IO:
  - for magnetic disks, random IO is very expensive compared to sequential IO

    - for SSD disks, random IO is still expensive but not as much as magnetic disks
    - avoid random IO to minimize delay
- **buffers** and buffer pools can be used to help hide disk latency:
    - temporary main memory cache for disk blocks
        - avoids future reads for "hot" ie. frequently-accessed data
    - most DBMS let users change buffer pool size

Databases on Disk

- how do we store database tables into disks?
    - just concatenate the tables back to back on disk
    - but the block size will rarely be a multiple of the tuple size:
        - leaves left over space at the end of each block
        - in an **unspanned** approach, leave the remaining space empty:
            - in the worst case, will waste just less than 50% of the block
            - typically chosen approach
        - in a **spanned** approach, place part of the last tuple into the remaining space and have it overlap into the next block:
            - optimal memory space
            - but when accessing certain tuples, two blocks will have to be read
- however, tuple sizes may vary eg. with `varchar` :
    - one approach is to reserve the maximum space for each tuple
        - much easier updates, but less optimal space usage
    - another approach is to use variable-length space for tuples:
        - packs tuples tightly
        - to detect the end of a tuple, have to either store the length as a metadata or a marker for the end of a tuple
        - to delete or update a tuple, may have to compact and reorganize the tuples in the block if the tuple has changed in size
        - how do we point to a tuple if they have nonconsistent lengths?
            - use a header that points to the different tuples in a **slotted page** approach
            - the location of the pointers in the header never changes, so the internal structure can be reorganized independently
- handling very long tuples that are larger than a block:
    - eg. `varchar` as well as BLOB and CLOB types used to store images, videos, or audios can be very large
        - **binary large object (BLOB)** and **character large object (CLOB)**
    - one approach is to simply use multiple sequential blocks to store tuples

        * but it is unlikely we will query on these very large fields
    - thus, instead we can *split* the tuples so that long attributes are stored separately
        * maintain pointers to the separate blocks where the actual large attributes are stored
- for certain analytical tables, reading the entire row of a tuple may not be necessary:
    - eg. selecting all students at a certain GPA threshold
    - typical **row-oriented storage** forces us to read the entire row even if most columns are not needed for query processing
    - in **column-oriented storage**, store by column instead of row
    - *pros*:
        * much better compression and caching behavior
        * unnecessary columns can be skipped for query processing
    - *cons*:
        * column values of matching rows must be joined
        * insertion and updating of a row is more expensive, require multiple IOs
    - for general record keeping, want to improve performance on writes, would use row-oriented storage
        * while if more analytical queries were to be performed, column-oriented storage may be preferable
- **sequential files** are used such that the tuples are ordered by certain attributes in the blocks:
    - to insert a new tuple, we can either rearrange all the tuples or use a linked list implementation
    - however, if we are trying to insert a new tuple into an ordered block that is full:
        * need to add a header to point to potentially multiple overflow pages:
            · could move the tuple to be inserted, or the tuple that would be shifted to the next block
            · pros and cons for insertion vs. query complexity
        * can reserve some free space as well to avoid overflow

Database Indexing

- how do we *efficiently* query on data?
    1. naively, scan the entire database on disk to find matching tuples
        - extremely slow, doesn't scale
    2. if the tuples are sorted, we can perform a binary search on the database:

    - $log_2 n$ complexity, but have to perform many disk IOs
    - a single disk IO takes approximately 10ms, which can add up when many users are querying the database
  3. build an **index** on the table:
    - an *auxiliary* structure to help us quickly locate a tuple given a search key
- in a **dense primary index**:
  - in a primary ie. clustering index, we index based on the primary search key
  - in a dense index, create a `(key, pointer)` pair for *every* record
    * can take up significantly space
  - find the key from the index and follow the pointer
    * eg. binary search or hash table
  - may still be using binary search on the same number of entries:
    * however, the dense index is much smaller and would require fewer disk IOs to search on
    * may even be able to fit into main memory
  - note that the index must be maintained across database updates
- in a **sparse primary index**:
  - create a `(key, pointer)` pair for every *block*, pointing to the first tuple in the block
  - *pros*:
    * less space required
    * if this sparser index can fit into memory, only requires one disk IO
- in a **multi-level index**:
  - have a dense first level and a sparse second level index
    * note that the sparse second level only has to refer to each block the first level *index* is stored in
  - *pros*:
    * essentially *guarantees* that the second level will be able to cache entirely in memory
      · requires always only two disk IOs
  - *cons*:
    * more space required for both levels than dense or sparse index alone
  - note that a dense second level index would defeat the purpose of conserving space
- in a **secondary non-clustering index**:
  - tuples in the table are *not* ordered by the index search key
    * eg. could be an unordered file or indexing on a non-search key in a sequential file
  - note that the first level index *must* be always dense in order to remap the tuples in order by search key

        * can be sparse from second level onwards
- considerations of the **indexed sequential access method (ISAM)**:
    - a multi-level index is actually quite performant when the underlying data is static
        * simple and efficient
    - however, when the underlying dataset is dynamic and requires frequent updates:
        * need to reserve overflow blocks for the actual blocks on disk *as well as* the indexes
        * with more and more updates, searches and organization become much less efficient wtih many overflow blocks at the data and index level
    - *pros*:
        * simple
        * sequential blocks
    - *cons*:
        * not suitable for updates
        * loses sequentiality and balance over time

B+ Trees

- the **B+ tree** is the most popular index structure in RDBMS:
    - *pros*:
        * suitable for dynamic updates
        * *always* balanced, so there is a consistent number of disk IOs done for queries
            · typically 1 or 2 IOs, at most 3
        * minimum space usage guarantee
    - *cons*:
        * non-sequential index blocks
- a B+ tree is a balanced tree:
    - starting from the root, the height of the tree is the same regardless of the path followed
        * ie. all leaf nodes are at the same level
    - every node corresponds to one disk block
        * typically a node will hold thousands of pointers to other tree nodes
    - designed for an index stored on *disk*
        * ie. disk-resident data structure, though some nodes will be cached in memory
    - for all nodes
        * if n pointers are stored, $n - 1$ keys are stored

- for leaf nodes:
  - all but the last pointer records the key of the tuple it is pointing to
  - the last pointer points to the next neighbor
  - at least half the pointer spaces are used, specifically at least $\lceil \frac{n+1}{2} \rceil$ pointers
    - note that for the rightmost leaf node, its neighbor pointer is considered used even though it is null
- for non-leaf nodes:
  - pointers point the nodes one level below, rather than tuples:
    - points to the tuples delimited by neighboring keys
    - eg. the pointer between key 23 and 56 points to the node referring to tuples in the range $[23, 56)$
    - eg. the leftmost pointer with key 23 to its right refers to tuples that are < 23
  - at least half the pointer spaces are used, specifically at least $\lceil \frac{n}{2} \rceil$ pointers
- for the root
  - always has at least 2 pointers and 1 key
- ex. Determine n given a 1024 byte node, a 10 byte key, and an 8 byte pointer.
  - $8n + 10(n-1) \leq 1024$ so $n = 57$

- search on B+ tree:
  - find a greater key and follow the link on the left
  - to search within a node's pointers, may do binary search or sequential search
    - the in-memory search does not dominate the overall search complexity as per Amdahl's law
  - note that the B+ tree also facilitates easier range searches
    - once at the leaf level, one leaf node will point to the next, just have to iterate through
- inserting on B+ tree:
  1. in the case of no overflow:
     - run the search algorithm
     - insert the key along with a pointer to the tuple to be inserted in the open space of the leaf
  2. in the case of leaf overflow:
     - split the leaf in two, put half of the keys in each leaf
       - note this validates the condition that nodes are at least half full because we only split when a node is full
     - *copy* the first key of the new node and insert that key together with a pointer to the new node to its parent
     - if non-leaf node doesn't overflow, we are done
  3. in the case of non-leaf overflow:

    - split the overflowing node into two
    - *move* the middle key up to its parent:
      * maintains nodes at least half full
      * this step can recursively introduce overflow at the parent
  4. in the case of root overflow:
    - middle key on overflowing root moves up to become a new root
    - again maintains at least two pointers on the root
- deleting on B+ tree:
  - need to check the minimum space guarantees
  1. in the case of no underflow, we are done after removing the tuple from the leaf node
  2. in the case of leaf underflow and we can coalesce with a neighbor:
    - check left and right neighbors, if either has space to merge the current node, then we merge:
      * once everything is removed, *delete* the pointer and key to the empty leaf node from its parent
      * this is the inverse of the copying step done in leaf overflow
    - check underflow on the parent
  3. otherwise, there is no space to merge the current leaf, so we redistribute tuples from a neighboring leaf:
    - ie. *move* from a neighboring leaf so both leaves are roughly half full
    - need to update a key in the parent
      * parent will not underflow since its number of pointers does not change
  4. in the case of non-leaf underflow and we can coalesce with a neighbor:
    - again, try to merge with a neighboring node:
      * when merging a non-leaf node, need to *pull* down the middle key from the parent into the merged node
      * this is the inverse of the moving step done in non-leaf overflow
    - check underflow on the parent
  5. otherwise, there is no space to merge the current non-leaf, so we redistribute non-leaf nodes:
    - perform a temporary overflow into one node by again pulling in the parent's middle key
    - then, *re-pick* the new middle key to move back up, and then split the two non-leaf nodes
      * ie. performing the non-leaf overflow algorithm again
  6. in the case of root underflow:
    - root only holds two pointers at minimum, so it will only underflow when it afterwards becomes empty
    - just remove the empty root node
    - reduces the tree height by 1

- index creation in SQL:
  - MySQL by default indexes on the primary key and unique attributes
  - eg. `CREATE INDEX sid_idx ON Student(sid)` creates a B+ tree on the specified attributes:
    - speeds up lookup on the `sid`
    - used to create index on an attribute that is neither the primary key or a unique attribute

Hash Indexing

- a **hash table** stores key-value pairs with constant lookup:
  - using a **hash function** that maps a key to an index $h(k) : k \to [0, \dots, n]$
    - a form of indirection
  - with an array $T[0, \dots, n]$, given a key k, store the associated value in $T[h(k)]$
    - if the array is sufficiently large, we can reduce the probability of collisions as well
  - allows for constant lookup
  - can we use the same idea to store tuples by their keys into a large hash table made up of blocks?
    - storing tuples on disk instead of memory, so we need to use blocks
    - since blocks are large, each block can hold multiple tuples whose keys are mapped to the same index by the hashing function
      - ie. each block corresponds to a hash bucket
  - however, if the hash algorithm is not uniformly distributed, we will have poor performance with any hash table due to collisions
- in addition, a secondary index can be used with DBMS hash tables:
  - store a different key together with a pointer to the matching hash bucket
  - secondary index may be hashed or unhashed
- issues with overflow and chaining:
  - if a block is full when we are inserting a tuple, need to add overflow buckets to spill over tuples into
    - add a pointer to the overflow bucket
  - as data grows in size, overflow blocks become unavoidable:
    - however, with many overflow chains to a block, performance degrades significantly
    - no longer constant lookup, lookup becomes potentially unbounded
- key concepts behind **extendable hashing**:
  1. only examine i of the b bits from the hash output, increasing i as needed
     - i starts at 0 ie. none of the hashed key is used to index
  2. add an additional level of indirection:

    - i bits of the hashed key are used to index into a directory holding pointers to hash buckets
    - key no longer indexes directly to the hash buckets
  - directories and buckets will each track their current i value:
    * the i-value on a directory indicates how many bits should be used to index into it
    * the i-value on a bucket indicates how many bits all tuples in the same bucket share in common
  - initially, $i = 0$ everywhere for empty tables:
    * none of the hashed key is used to place tuples into buckets and directories
    * directory has a single entry pointing to one empty bucket
      · no need to check directory, everything can be stored in the bucket
- inserting a tuple into an extendable hash table:
  1. index into the directory based on the initial i prefix of the hashed key
  2. follow the pointer and insert the tuple into the hash bucket
  - however, when the bucket to insert into becomes full, we need to split the bucket in two and update the directory:
    * to split the bucket in two, we will need to allocate a new block ie. bucket
      · but now we have to update the directory, which may be full
    1. when the i-value of the bucket and the i-value of the directory is the same, there is no more space in the directory:
      * need to first *double* the size of the directory for the new bucket
      * copy the directory over and extend all indices by an additional bit
        · ex. the old directory indices `0, 1` become `0b00, 0b10` and the new directory indices are `0b01, 0b11`
      * copy the existing bucket pointers to its new immediate neighbors
        · ie. at this point, neighboring directory entries point to the same buckets
      * increment the i-value of the directory by 1
    2. create a new bucket and *redistribute* the tuples among the buckets based on the incremented i-value:
      * reroute necessary directory pointers to the new bucket
      * increment the i-values of the buckets by 1
    * note that splitting the bucket may still leave to overflow if all the tuples still redistribute into the same bucket
      · have to rerun the insertion algorithm recursively until there is no more overflow
  - this solves the dynamic overflow chain since the lookup is always con-

    stant
        * if directory caches into memory, only have to perform a single disk IO, otherwise will require two disk IOs

- deleting a tuple from an extendable hash table:
    1. search for a key
    2. remove the tuple
    3. merge buckets if possible:
        - if buckets are adjacent, have the *same* i-values, and can fit their contents into a single bucket, they can be merged together
            * same i-values indicates they have the same split level, and adjacency means they have matching prefixes
        - combine the buckets together, and decrease the i-values
        - update the directory pointer for the merged buckets
    4. shrink directory if possible:
        - whenever neighboring directory entries all point to the same bucket, we can shrink the directory
        - equivalently, when all bucket i-values are smaller than the directory i-value
- is there a minimum space guarantee for extendable hash tables?
    - unlike in B+ trees, there is no flexibility of where to split:
        * depends on the matching prefixes of hashed keys in a bucket
        * cannot choose to always promote the middle key like in B+ trees
    - need a good hash function that distributes the hash prefixes well
- comparing with B+ tree:
    - for a range query such as `SELECT * FROM R WHERE R.A>5` :
        * an extendable hash would not work well, since the hash function will typically not preserve the ordering or range properties of the original key
            · would have to recalculate hashes and index into the table again
        * while in a B+ tree, the range query does well since the leaves can be traversed in order by following the tail pointers
    - for a point query such as `SELECT * FROM R WHERE R.A=5` :
        * extendable hash excels at point searches with a constant access cost of at most 2 disk IOs
        * B+ tree still does well on the order of the height of the tree
    - the efficiency of a B+ tree is quite competitive when we consider that the fanout factor is typically very large:
        * the depth of the tree is roughly log_n where n is the fanout factor:
            · fanout is typically on the order of hundred or thousand
            · thus typical tree depth is under 3
        * with a relatively large block size, the fanout factor is quite large
        * eg. to store 1 million tuples with a fanout of 1000, there are only two levels in the tree to search through with 1000 leaves

    - since the root will cache into main memory, this leads to roughly equivalent performance to the extendable hash of 2 disk IOs
  - thus, B+ trees are the typical choice for a default indexing structure since they have good all-around performance

Joins

- how do we implement a join $R \bowtie S$ in a database?
- in the **nested loop join (NLJ)** algorithm:

```
1. for each $r \in R$:
     – for each $s \in S$:
         * if $r.A = s.A$, output $(r, s)$
```

  - perform a double for-loop on the two tables
  - compare each element of R against each element of S:
    - in a join, must at least iterate over all tuples of one of the tables eg. R
    - requiring an *additional* full scan of the other table eg. S for every tuple of the first table is the most expensive operation of the algorithm
  - whenever the matching attributes equal one another, output the merged tuple
  - this is the original algorithm used by MySQL
- in an **index join (IJ)**:

```
1. create an index on $S$ for the joining attribute if necessary
2. for each $r \in R$:
     – lookup the index of $S$ for the joining attribute of $r$
     – output any matches from the index
```

  - avoids many unecessary extra scans on S
    - may be worth the additional overhead of creating a new index
- in a **sort-merge join (SMJ)**:

```
1. sort the relations first
2. perform a join:
     – this sorted join can be performed in linear time based on size of each table
     – like the linear merge algorithm, maintain pointers to each table:
         * compare and increment the pointer to the tuple with the smaller value, without ever moving pointers backward
         * if matching, output matching tuples and increment both pointers
```

- in a **hash join (HJ)**:
  - given a hash function h, can two tuples r, s join if $h(r.A) \neq h(s.A)$?

    * no, they cannot join since the joining attribute *must* be different

1. partition both tables based on the hash values of the joining attribute into buckets
2. join tuples in matching buckets
    - tuples in different buckets *cannot* join with each other
    - significantly reduces the number of comparisons to make

Algorithm Comparisons

- how do we judge performance of the different join algorithms?
    - need a **cost model** to estimate the performance of a join algorithm:
        * eg. one model is the number of disk blocks read or written during a join, ignoring the last IO used for writing the final results
            · good estimate since join cost is dominated by disk IO
        * different models have different drawbacks, eg. number of disk blocks ignores random or sequential IO differences, CPU cost, etc.
- we will compare the join algorithms on a disk IO cost model, with the following example system information:
    - two tables R, S
        * $|R| = 1000, |S| = 10000, |R| < |S|$
    - 10 tuples per block, so $b_R = 100$ blocks and $b_S = 1000$ blocks
    - M or main memory can cache up to 22 disk blocks
- cost of sort-merge join:
    - in the join stage, after tables are sorted, we have two pointers to each table:
        * however, when reading and writing, must be done in a block rather than a tuple granularity
            · need to stage the output in a cache block as well until it can be written out to disk (IO cost will be ignored in cost model)
        * at any time of the merge, we are using 3 blocks in memory
            · one to read a block of R, one to read a block of S, and one to write the output
        * when one block is exhausted ie. the cursor reaches the end of the block, read the next block from disk
        * altogether, the algorithm reads $b_R + b_S$ blocks or $1000 + 100 = 1100$ blocks in our example
            · each block is read once
        * but we are only using 3 blocks of memory, could we make use of more memory?
            · eg. memory has enough space read 10 blocks at a time instead of just 1

        · in our cost model that ignores sequential vs. random IO, the same number of blocks is still read from disk
      * this is the ideal estimation, given tables are already sorted on disk
  – to actually sort the tables, the cost of disk IOs is $2b_R(\lceil log_{M-1}\frac{b_R}{M}\rceil + 1)$
  – altogether, the disk cost for sort-merge join is:

$$2b_R(\lceil log_{M-1}\frac{b_R}{M}\rceil + 1) + (b_R + b_S)$$

      * 1100 disk IOs in the example setup if sorted, 7500 disk IOs if not sorted

- cost of nested loop join:
  – scan S once for every tuple of R
  – each block of R is read once
  – naively, we can read each block of S for every *tuple* of R
      * altogether, this version reads $b_R + |R| \times b_S$ blocks or $100 + 1000 \times 1000 = 1000100$ blocks in our example
  – we can do better by reading each block of S for each *block* of R in a **block nested loop join**:
      * ie. compare matching tuples on a block granularity, since the entire block is cached into memory already
      * altogether, this version reads $b_R + b_R \times b_S$ blocks or $100 + 100 \times 1000 = 100100$ blocks in our example
        · reduces cost by an order of magnitude
  – however, we can do even better by using the remaining blocks in memory:
      * ie. read *more* blocks of R and compare all these blocks at once with blocks of S
      * at a maximum, we can read in at most 20 blocks into memory, since one block is used for output staging and another for reading the blocks of S
  – altogether, the disk cost for the best version of nested loop join, the block-nested loop join, is:

$$b_R + \lceil\frac{b_R}{M-2}\rceil b_S$$

      * $100 + \lceil\frac{100}{20}\rceil 1000 = 5100$ disk IOs in the example setup
        · only factor of 5 difference from ideal sort-merge
      * note that if we scan over R for each tuple of S, we would read $1000 + \lceil\frac{1000}{20}\rceil = 6000$ blocks in our example
        · generally want to scan over the larger table

- cost of hash join:
  – partitioning stage:
      * need to read each table and hash them into k buckets

      - note that the staging for outputs requires k blocks as buffers for the outputted buckets
    - thus we can only generate $M - 1$ partitions at a time
      - final block is needed to read from the table
    - to write out the buckets:
      - assuming perfect hashing, this requires $\lceil \frac{\textit{blocks}}{M-1} \rceil (M-1)$ disk IOs
      - in practice approximately $2 \times$ *blocks* disk IOs, since the output buckets have approximately the same number of blocks that are read in
    - in general, $2b_R + 2b_S$ disk IOs are performed or $2000 + 200 + 2200$ disk IOs in the example setup
  - joining stage:
    - under the best-case assumption that each bucket of the smaller table fits in main memory:
      - we can put all blocks for a bucket in main memory, and then scan opposite bucket in the other table block by block
      - need one block for opposite bucket, another block for writing the result, and the rest of the memory blocks can be used for loading in all blocks of a bucket
    - have to read approximately $b_R + b_S$
  - altogether, the disk cost for hash join is:

$$2(b_R + b_S) + (b_R + b_S) = 3(b_R + b_S)$$

  - $3 \times 1100 = 3300$ disk IOs in the example setup
    - generally, each block is read and written in partitioning stage, and then read in joining stage
  - what if the blocks for a bucket do not fit into main memory?
    - note that joining between buckets reduces down to *another* join problem!
      - can *rebucket* to create new sub-buckets
    - ie. recursively partitioning the bucket using a new hash function into even smaller partitions
      - note that we must use a *different* hash function than the one originally used, otherwise all keys will map to the same bucket again
    - the number of recursive bucketizing steps needed for R is $\lceil log_{M-1} \frac{b_R}{M-2} \rceil$:
      - bucketizing can stop when the size of the bucket is $\leq M - 2$ blocks
      - after a single bucketizing step, we have $\frac{b_R}{M-1}$ blocks in each bucket
      - after n bucketizing steps, we have $\frac{b_R}{(M-1)^n}$ blocks in each bucket

- then, the overall disk cost for hash join with rebucketizations is

$$2(b_R + b_S)\lceil log_{M-1}\frac{b_R}{M-2}\rceil + (b_R + b_S)$$

- cost of index join:
  - overall, the disk cost for index join is:

    $$b_R + |R|(C_I + C_S)$$

    * C_I is the disk cost for the index lookup
    * C_S is the disk cost for matching the S tuples with R tuples
    * in the example setup, requires between 1115 and 10640 disk IOs depending on the size of the index and other constraints
  - if 15 blocks are used for the index (1 root, 14 leaves):
    * can fit entirely into main memory along with one block for reading R, another for reading S, and another for writing out the result
    * on average, assume every R tuple on average has one matching S tuple

    1. disk cost for the R scan is 100 disk IOs
    2. disk cost for the index lookup occurs just once, and requires 15 disk IOs
    3. disk cost for reading the matching S tuples in the worst case is 1000 disk IOs, one for each tuple of R
  - can we improve this?
    * always performant to cache the indexes, saves 2 disk IOs on every read of R to lookup matching S tuple
    * can we use remaining blocks in memory to cache extra blocks of S?
      · but assuming S is stored randomly with respect to the joining attribute, if we store 4 extra blocks of S in the remaining memory slots, the saved disk IOs is only $\frac{5}{1000}$
      · not much increased performance when caching tables in an index join
  - if 40 blocks are used for the index (1 root, 39 leaves):
    * in addition, assume every R tuple on average matches with 10 S tuples
    * want to cache the index, but it no longer fits into memory completely:
      · always cache the root, and cache as much of the leaves as possible
      · thus we can cache 18 leaves, leaving 21 leaves uncached

    1. disk cost for the R scan is still 100 disk IOs
    2. disk cost for the index lookup requires $1 + 18 + 1000 \times 1 \times \frac{21}{39}$ disk IOs on average

        - 19 blocks are initially cached, remaining blocks may have to be accessed during index lookup
    3. disk cost for reading the matching tuples is 1000×2 disk IOs if S is clustered over the joining attribute or 1000×10 disk IOs if S is not clustered:
        - if clustered, reading the 10 matching tuples will require at worst reading two adjacent blocks
        - otherwise, may have to read each of the 10 matching tuples from a different block

Table 16: Comparison of Algorithms

Join Algorithm	Formula ($b_R < b_S$)
Nested Loop Join	$b_R + \lceil \frac{b_R}{M-2} \rceil b_S$
Sort-Merge Join	$2b_R(\lceil log_{M-1} \frac{b_R}{M} \rceil + 1) + (b_R + b_S)$
Hash Join	$2(b_R + b_S)\lceil log_{M-1} \frac{b_R}{M-2} \rceil + (b_R + b_S)$
Index Join	$b_R + \lvert R \rvert (C_I + C_S)$

- comparison conclusion:
    - when the table sizes are reasonably small, NLJ is good enough
        - comparable to the other algorithms
    - hash join is usually the best for joins based on equality ie. equi-joins:
        - at least if tables have not been sorted and there is no index
        - consider merge join if tables have beens sorted
        - consider index join if index exists
    - to pick the best, DBMS needs to maintain data statistics

Query Optimization

Ex. Query optimization:

```
SELECT * FROM R, S, T
WHERE R.B=S.B AND S.C=T.C AND R.A=10 AND T.D<30;
```

- how can we process the above query ie. set up a **query execution plan**?
    1. join R and S, then join with T, and then filter based on each condition subsequently
    2. filter on T, then join with S, then join with R, then filter on R
    - many different possible plans, etc.
        - for the joining alone, there are 12 different ways to join the tables
            - $3! = 6$ orderings, and then 2 ways to join an ordering of two tables

        * in general, for n way joins, there are $\frac{(2(n-1))!}{(n-1)!}$ possible execution plans
        * note that $R \bowtie S \neq S \bowtie R$
    - in reality, picking the very best plan is too difficult
    - DBMS tries to avoid obvious mistakes using a number of heuristics to examine only those plans that are likely to be good:
        * put smallest table on the left
        * left-deep trees
        * push down selection conditions as deep as possible
        * etc.
    - thus the DBMS will generally pick a good pattern of execution eg. 90% of the time
    - many systems allow users to examine the query plan:
        * no SQL standard, different syntaxes per system
        * eg. `EXPLAIN SELECT` in MySQL and PostgreSQL
        * eg. `EXPLAIN PLAN FOR SELECT` in Oracle
        * eg. `SET SHOWPLAN_TEXT ON` in MS SQL Server
- DBMS uses statistics on tables and indexes to pick the best query execution plan:
    - ie. uses a cost-based optimizer
    - eg. in Oracle, can collect statistics through
        * `ANALYZE TABLE <table> [COMPUTE | ESTIMATE] STATISTICS`
    - eg. in DB2, `RUN ON TABLE <userid>.<table> AND INDEXES ALL`
    - eg. in MySQL does not have a cost-based optimizer
        * instead is a rule-based optimizer that uses simple heuristics

Transactions

- a **transaction** is a sequence of SQL statements that are executed as a single unit:
    - DBMS guarantees the **ACID** property on all transactions
    1. atomicity ie. all or nothing:
        - either all or none of the operations in a transaction is executed
        - if system crashes in the middle of a transaction, all changes are undone
    2. consistency if the database was in a consistent state before transaction, it is still in a consistent state afterwards
    3. isolation
        - even if multiple transactions run concurrently, the final result is the same as each transaction runs in isolation sequentially
    4. durability

    - all changes made by committed transactions remain even after system crash
- transactions in SQL:
    - after a `COMMIT`, all changes made by the transaction are stored permanently
    - a `ROLLBACK` undoes all changes made by the transaction
    - in `AUTOCOMMIT` mode, every SQL statement becomes one transaction
        * otherwise, all SQL commands through commit and rollback become one transaction
    - setting autocommit mode:
        * `SET AUTOCOMMIT ON/OFF` in Oracle, default off
        * `SET AUTOCOMMIT = {0|1}` in MySQL, default on
        * `SET IMPLICIT_TRANSACTIONS OFF/ON` in MS SQL server, default off
        * in Oracle, MySQL, and MS SQL, `BEGIN TRANSACTION` temporarily disables autocmmit mode until commit or rollback
    - a transaction can be declared to be read only when it has `SELECT` statements only
        * DBMS can use this information to optimize for more concurrency
- by default DBDMS guarantees ACID for transactions:
    - some applications may not require ACID and may want to allow minor "bad" scenarios to gain more concurrency
    - by specifying the **SQL isolation level**, developers can specify which exceptions can be allowed
        * eg. dirty read, non-repeatable read, phantom
- in a **dirty read**, one thread updates the DB while the other is reading:
    - should the reading thread see the updated value?
    - eg. T1 executes `UPDATE Employee SET salary=salary+100` and T2 executes `SELECT salary FROM Employee where name='Amy'`
    - dirty read may be acceptable in some applications
        * among the 4 SQL isolation levels, `READ UNCOMMITTED` allows for dirty reads
- in a **non-repeatable read**, a thread reads the same tuple multiple times, it may read a different value if the tuple has been updated:
    - eg. T1 executes `UPDATE Employee SET salary=salary+100 WHERE name='John'` and T2 repeatedly executes `SELECT salary FROM Employee WHERE name='John'`
    - the SQL isolation levels `READ UNCOMMITTED` and `READ COMMITTED` allow for non-repeatable reads
- in **phantom reads**, when new tuples are inserted, statements may or may not see part of them:
    - preventing phantom reads can be very costly
        * requires an exclusive lock on the entire table or a range of tuples
    - except the isolation level `SERIALIZABLE`, phantoms are allowed

Table 17: SQL Isolation Levels

	Dirty Read	Non-Repeatable Read	Phantom
`READ UNCOMMITTED`	Y	Y	Y
`READ COMMITTED`	N	Y	Y
`REPEATABLE READ`	N	N	Y
`SERIALIZABLE`	N	N	N

- declaring SQL isolation level:
  - `SET TRANSACTION [READ ONLY] ISOLATION LEVEL <level>`
    - `READ UNCOMMITTED` is allowed only for the `READ ONLY` access mode
  - `READ COMMITTED` is default in Oracle and MS SQL server
  - `REPEATABLE READ` is default in MySQL and IBM DB2
  - isolation level needs to be set before every transaction