SQL Queries

```
WITH table1 AS (...),
     table2 AS (...)
  SELECT DISTINCT col1, col2 alter_name
   WHERE col3="A" AND (col4 IS NOT NULL)
    FROM table_1 t1 JOIN (...) t2
      ON t1.id = t2.id
GROUP BY col5
  HAVING col6 > 1
ORDER BY COUNT(*) DESC -- default: ASC
-- Aggregation Functions:
--    COUNT, SUM, AVG, MIN, MAX

UNION --/EXCEPT/INTERSECT

SELECT col    FROM table1 WHERE id IN (
SELECT t1_id FROM table2 WHERE A>1);
```

CHECK works on single columns:

```
CREATE TABLE Album ...
   CHECK (NumSongs = (SELECT COUNT(*) FROM Songs s
                         WHERE s.SingerID=Album.SingerID AND
s.AlbumName=Album.AlbumName))
```

Update, Insert, Delete Commands and Useful Func.s

```
INSERT INTO table_name( Column1, Column2, ... )
       VALUES ( 'Value1', 'Value2', ... );
DELETE FROM table_name WHERE some_col=some_value;
UPDATE movies SET invoice='paid' WHERE paid > 0;
```

`COALESCE(col1, col2, "default value"):` Return the first non-NULL argument.

DDL↓

```
CREATE TABLE People (
  name VARCHAR(9) not null,
  ssn  INTEGER     primary key, -- implies UNIQUE
  col1 LONG        references foreign_table(key),
  male BOOLEAN     default true,
  role VARCHAR(9) check (role in ('user','admin')
))); -- or:
-- FOREIGN KEY (c1, c2) REFERENCES table2(k1, k2)
-- PRIMARY KEY (ssn, name, id)    --VIEWS:
CREATE OR REPLACE VIEW view1 AS
SELECT patient_id, patient_name from patients;
```

DDL for Weak Entity ↓

```
Create table Book(
   ISBN varchar(30), Title varchar(40), Primary key (ISBN));
Create table BookCopy(
   ISBN varchar(30), copy# varchar(20),
   Primary key(copy#, ISBN),
   Foreign key ISBN refers Book(ISBN) ON DELETE CASCADE));
```

What is the result of the following query?

`select 2+2 as Num from Person;` ANSWER:
The schema of the result relation is (Num). The result is a table of n tuples of value 4. Note that if Person is empty then so is the result table (n=0).

Behavior of NULL

- An **arithmetic operation** involving a NULL returns NULL. For example, NULL - NULL = NULL, not zero.
- A **boolean comparison** between two values involving a NULL returns neither true nor false, but unknown in SQL's three-valued logic. E.g., neither NULL = NULL nor NULL != NULL is true. To see if a value is NULL, use IS NULL.
- An SQL query **selects** only values whose WHERE expression evaluates to true, and groups whose HAVING clause evaluates to true.
- The **aggregate** COUNT(*) counts all NULL and non-NULL tuples; COUNT(attribute) counts all tuples whose attribute value is not NULL. Other SQL agg. func. ignore NULL values in their computation.
- MAX() of NULLs only: returns NULL.

Six basic operations

- Projection $\pi_\alpha(R)$
- Selection $\sigma_\theta(R)$
- Union $R_1 \cup R_2$
- Difference $R_1 - R_2$
- Product $R_1 \times R_2$
- (Rename) $\rho_{\alpha\to\beta}(R)$

And some other use

- Join $R_1 \bowtie_\theta R_2$
- Semijoin $R_1 sj_\theta R_2$
- Intersection $R_1 \cap R_2$
- Division $R_1 \div R_2$

	Heap File	Sorted File	Hashed File
Scan all recs	p(T) D	p(T) D	1.25 p(T) D
Equality Search	p(T) D / 2	D $\log_2$ p(T)	D
Range Search	p(T) D	D $\log_2$ p(T) + (# pages with matches)	1.25 p(T) D
Insert	2D	Search + p(T) D	2D
Delete	Search + D	Search + p(T) D	2D

Keys: A set of ≥1 columns.

Superkey: A key that is a superset to a candidate key. (Therefore, a superkey must contain >1 columns.)

Minimal Super key ≡ **Candidate Key**: A key that can uniquely identify each row in a table.

Primary Key: The chosen Candidate Key for doing that.

Secondary key / Alternate key: A Candidate Key not chosen for doing that.

Search Key: A key used for locating records. **Sort** or **control key**: A key used to physically sort the stored data.

Composite key or **concatenate key:** A key with >1 columns. (Usually implies "composite primary key".)

Foreign Key: A key in one table ("the dependent table") that matches the PK of another table ("the parent table").

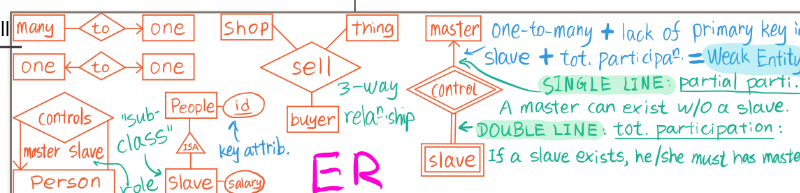
Armstrong's Axioms — Properties of FD

- Reflexivity: $Y \subseteq X \Rightarrow X \to Y$
- Augmentation: $X \to Y \Rightarrow XZ \to YZ$
- Transitivity: $X \to Y + Y \to Z \Rightarrow X \to Z$
- **Union**: $X \to Y + X \to Z \Rightarrow X \to YZ$
- **Decomposition**: $X \to YZ \Rightarrow X \to Y + X \to Z$
- **Pseudotrans.**: $X \to Y + WY \to Z \Rightarrow WX \to Z$

Types of Dependencies

From the full key: full PK → outside of the PK
Partial dependency: part of the PK → outside
Transitive dependency: outside → outside
Into-key dependency: outside → into the PK

First Normal Form (1NF): there are a fixed number of columns.
Second Normal Form (2NF): 1NF and no partial dependencies
Third Normal Form (3NF): 2NF and no transitive dependencies.
Boyce-Codd Normal Form (BCNF): 1NF + all dep. from full



$\{R_1, R_2\}$ is a lossless decomposition of $R \Leftrightarrow R_1 \cap R_2 \to R_1 - R_2$ or $R_2 - R_1$

If fails: Create a new sub-relation of $R_1 \cap R_2 + (R_1 - R_2)$ or $(R_2 - R_1)$.

$\{R_1, \ldots, R_n\}$ is dependency preserving if $F^+ = (F_{R_1} \cup \ldots \cup F_{R_n})^+$.

Quick Validation: For $\forall$ A→B in F^+ of R, $\exists$ R that contains A, B.

No Redundancy / BCNF

- not always compatible w/ FD preservation:

Validation: Every FD in F+ should come from a superkey (including PK) of some sub-relation.

Algorithm: (right). If assertion fails, try 3NF:

1. Given F, calc its min cover Fm.
2. For every X->Y in Fm, create a relation Ri=XY.
3. If the key of R, K, is not in any R, create a relation Ri=K. 3NF preserves FDs.

```
def decomp(R):
  for A->B in R0.F+.proj(R):
    if A is not a superkey of R:
      new R1 = A+B
      new R2 = A+(R-B)
      return decomp(R1), decomp(R2)
  return R # If R is fine.
decomp(R0)
assert this is FD-Preserving
```

←**Decompose** **Isolation Levels↓** **ER Diagrams ↑**

	Serializabl.	Repeatabl.R.	R.Commited	R.Uncomm.
Rel. X-locks	At the end	At the end	At the end	Not obtained
Rel. S-locks	At the end	At the end	ASAP	Not obtained
Rel. Range L.	At the end	Not obtained	Not obtained	Not obtained
Phantom Read	Prevented	May be	May be	May be
Nonrepeat.Rd.	Prevented	Prevented	May be	May be
Dirty Reads	Prevented	Prevented	Prevented	May be

Deadlocks

If multiple transactions acquire locks on data items in a specific order (a transaction doesn't have to acquire locks on each data item involved), then no cycle can happen

Closure of a FD set F, denoted by F+, is the set of all FDs logically implied by F.
Algorithm: Repeat these properties in any order on all FDs in F+, till no change:

How to find Minimal Cover of a FD set F: Repeat these operations in any order on any FD, till no change can be made: 1 Use the Decomposition Rule: X→YZ => X→Y, X→Z. 2 Simplify LHS: AB→C, A→B => A→C , A→B. 3 Simplify RHS: A→BC, A→B => A→C , A→B.

Closure of an attribute. a: a+ is the set of all attrib.s possibly can be determined by a through FDs in F. **Algorithm:** `result = set(a); while result.updated: for A->B in F: if A in result: result+=B.`

ACID

- **atomicity:** either all operations in a transaction are executed, or none is.
- **consistency:** each transaction is executed in isolation keeps the database in a consistent state (this is the responsibility of the user and constraints on the DB)
- **isolation:** transactions won't affect each other.
- **durability:** updates preserves!

Conflicts

WR Conflict (dirty read): A transaction T2 could read a database object A that has been modified by another transaction T1, which has not yet committed.

RW Conflict (unrepeatable read): T1 reads row A, A is 1. T2 then updates A to 2. Now if T1 reads row A again, A is now different. **(Phantom Read):** T1 queries X, {A, B} is returned. T2 adds row C that satisfies X. T1 query X later, {A, B, C} returned.

WW Conflict (lost update): A transaction T2 could overwrite the value of an object A, which has already been modified by a transaction T1, while T1 is still in progress.

Index Storage Alternatives

1. **Store actual rows with key values**
   - Consititutes as a file organization. It's a clustering and sparse index.
   - For each dataset, at most one index can be in Alternative 1.
   - Cheap for lookups; costly for insertions and deletions.
2. **(key, record id of matching data record)**
3. **(key, list of record ids of matching data records)**

For the last two: For each dataset, multiple indices can be in Alternative 2/3. A3 is more compact than A2, but leads to variable sized data entries even if search keys are of fixed length. Easier to maintain, but more costly to look up.

Conflict Serializable: Ways to see whether a schedule is *conflict serializable*:

- iff its precedence graph contains no cycles.
- iff it can be transformed into a serial schedule by swapping non-conflicting operations. Two operations are said to be conflicting if all conditions satisfy:
  ① They belong to different transaction ② They operation on same data item
  ③ At Least one of them is a WRITE operation
- Transactions that are all well-formed and 2PL are **serializable.**
  - **Well-formed**: Acquires locks as it should. (i.e., not `READ UNCOMM.`)
  - **2PL:** Don't acquire any lock after one lock has been released.

Types of File Organization

- **Heap files**: cheap for full-file scans & frequent updates.
  ○ data unordered ○ writes new data to the end
- **Sorted files**: cheap for sorted data retrieval & range searches.
  ○ need external sort or an index to keep sorted
- **Hashed files**: cheap for equality searches. ○ Collection of buckets with primary & overflow pages ○ hashing function over search key attributes

JOIN COSTS

[Merge Sort Join]
To sort each relation:
Number of Passes = $1 + \log_{B-1} \lceil N/B \rceil$ where B = # buffers, N = # pages.
SortCost: 2N * (# Passes).

Locks	2PL	S2PL	SS2PL
X-locks	Released ASAP	Released at the end	Released at the end
S-locks	Released ASAP	Released ASAP	Released at the end

[Hash Join] ≈ 3(b(R) + b(S)) **[Merge Join]** MergeCost: b(R) + b(S) + SortCost
[Block Nested Loop Join] b(R) + (b(R)/(B-2)) * b(S) **[Nested Loop Join]** b(R) + t(R)·b(S)
[Index Nested Loop Join] b(R) + t(R) * cost of matching in S (1.2 for hash index, 2-4 for B+ tree) and: - Clustered index: 1 I/O on avg. - Unclustered index: Up to 1 I/O per S tuple.

b(R): Number of blocks (pages of relation R) *r(R): Number of tuples of R (rows)*

Key Constraint: e.g. Any [Department] can only <be managed> by 0 or 1 [employee]: `[Dep.]→<manage>—[Emp.ee]`
Total/Partial Participation: e.g. ○ **Not all employees get to manage** — [Employees] partially part. in <manage>: *thin/single line*.
ER Features ○ **Each department must be managed —** [Departments] totally participates in <manage>: *thick/double line*.
Weak Entities: One-to-many relationship + Total Participation. **Each Weak Entity can only have one Owner Entity!**
○ **One-to-many relationship** — "the identifying relationship of the weak entity": `[The Owner] +——o< [The Weak]`
○ **Total Participation**: The weak entities must all ("totally") participate in the identifying relationship.
Class Hierarchies: Represented by a triangle with text "ISA". e.g.: There are 2 types of [Users]: [Free Users] and [Premium Users].
Aggregation: Represented by a dashed box surrounding a collection of entities + relationships. E.g.:
○ `[Employees]—<monitors>—{ [Projects]—<sponsored by>—[Departments]}` | ▪ A process contains these info:
- ▪ Zero, 1 or more employees can monitor the process. | - All projects must be sponsored by 1 or more departments.
- ▪ An employee can monitor 0, 1 or more processes. | - A department does not necessarily have to sponsor any project.

Classes of Indices
Primary / Secondary:
Primary has the Primary Key
Clustered / Unclustered:
Orderings of records and index are nearly the same.
Can cluster to only 1 index.
Dense / Sparse:
Dense has index entry per data item; Sparse may skip.
Alt.1 always gives Sparse.

Model for Analyzing Access Costs and I/O Costs
p(T): # of data pages in table T *r(T)*: # of records in table T *D*: average time to read/write a page
Seek Time: Cost switching tracks. ***Rotational Delay***: Cost to go to a sector when head is on right track. ***Page Transfer***: Cost reading a page

```
db.orders.mapReduce( () => { for (var item of this.items) { // map
      emit(item.name, {count: 1, qty: item.qty})}},
   (key, objs) => {                                     // reduce
     count: objs.reduce( (sum, obj) => sum + obj.count , 0),
     qty   : objs.reduce( (sum, obj) => sum + obj.qty   , 0)
   }, {out: {inline: 1}, query: {key: 'val'}, finalize:  // finalize
        (key, res) => {res.avg = res.qty/res.count; return res} });
```

! Print drinkers who like all of the beers that John likes.
```
WITH jb AS( SELECT  DISTINCT beer -- beers John likes
       FROM likes WHERE drinker='John')
SELECT DISTINCT l1.drinker -- select all people who
-- do not have a beer they don't like but John likes
FROM likes l1 WHERE NOT exist (
  SELECT beer -- any beer John likes
  FROM   jb   -- but not liked by this guy
  WHERE  beer NOT IN ( SELECT beer
    FROM  likes l2 -- all beers this guy likes
    WHERE  l2.drinker = l1.drinker ) )
```

Use Agg. Func.s in HAVING: For each drinker for whom there are >2 bars serving some beer they like, print the tot. # of beers they like that are served by some bar
```
SELECT l.drinker, Count(DISTINCT l.beer) AS total
FROM likes l inner join serves s ON l.beer=s.beer
GROUP BY l.drinker HAVING Count(DISTINCT s.bar)>2
```

Print All Cities That Have No Direct Flight to PHL
```
SELECT DISTINCT depart FROM flight
 F1 WHERE NOT EXISTS (SELECT *
FROM flight F2 WHERE F1.depart =
F2.depart AND F2.arrive = 'PHL')
```
OR
```
(SELECT depart FROM flight) MINUS (SELECT
depart FROM flight WHERE arrive='PHL');
```

Print all airports that have direct flights to all destinations that PHL does.
Lv. 3: Flights departing from a F1 city and arriving in a F2 city.
Lv. 2: Destinations of PHL flights, where no flight from a F1 city will arrive.
Lv. 1: Cities that does not arrive in a city described in Query Lv. 2.

```
SELECT depart FROM flight f1 -- this is where we select cities from
WHERE  NOT EXISTS (      -- these cities should not have:
  SELECT arrive FROM flight f2  -- a destin. city departed from PHL
  WHERE  depart='PHL' AND NOT EXISTS ( -- that doesn't have
    SELECT * FROM flight f3          -- going from
    WHERE f1.depart = f3.depart   -- "these cities" mentioned above
      AND f2.arrive = f3.arrive)) -- having same destination as
```

```
db.people.insert({name: 'Li'});
db.people.deleteMany({major: 'DATS'});
db.people.deleteOne ({name : 'Mary'});
db.old_Collection_Named_People.drop();
db.people.updateOne({_id: 1},
    {$set: {role: 'admin'}});
db.people.updateMany({_id: {$ne: 1} },
    {$set: {role: 'user'}} );
db.people.findOne().pretty();
db.people.find({
  "name.last"   : "Li",
  salary        : {$exists: true},
  tookCourses   : 'CIS550',//Array
  age           : {$gt: 20},
  gender: {$in  : ['M' , 'F' ]},
  skills: {$all : ["ML", "DB"]},
  $or           : [{majorIn: 'DATS'}
                   {worksAt: 'DATS'}],
 }, {name:true, _id:false}).sort(
    {score: 1 , age: -1   }
 ).limit(3).skip(1).count();
results = db.people.find().toArray();
db.people.find().forEach(
   (doc) => printjson(doc) );
db.demonstrate.aggregate([
  {$group: { _id: '$userID',
          total: {$sum: "$price"}} },
  {$sort : { dec: -1, asc: 1 }     },
  {$match: { sum: { $gte: 20,
                    $lte: 45 } }   },
  {$unwind : "$keyOfSomeArray"     },
  {$project: {toShow: 1, toHide:0} },
  { $limit : 5 }, { $skip : 5 }    ] );
db.business.mapReduce(
 () => emit(this.city, this.stars),
 (state, l) => {
  mean: l.reduce((p,c)=>p+c,0)/l.length,
  min : Math.min( ...arr ),
  max : Math.max( ...arr ) });
```

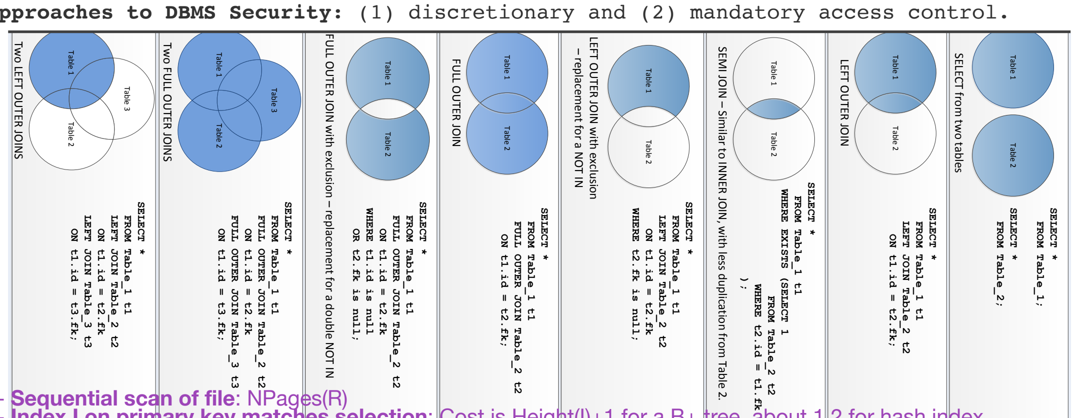
←MongoDB Neo4J↓
```
MATCH (m)<-[*1..5]-(n:Person {name: 'Alice'})-
 [rel:LOVES {since: 2010} | HATES]->(m:Person)
OPTIONAL MATCH p: (n)-->(m)   -- Assign a path to p.
    Optional pattern: nulls will be used for missing parts.
WHERE n.name = 'Alice' AND n.age > 10
WITH user, count(friend) AS numFriends
ORDER BY n.property DESC    SKIP 20 LIMIT 10
RETURN DISTINCT n AS columnName, type(m),
  collect(n.property) AS list_of_prop_vals;
- [*1..5]: any relation chain of 1~5 long.
- [*]:     any relation chain of any length.
- shortestPath    ( (n:Person)-[*..6]-(m) )
- allShortestPaths( (n:Person)-[*..6]-(m) )
```

```
- size( ... ) // Count the paths matching the pattern
CREATE (n:Person {name: $value})
SET n:Spouse:Parent:Employee
REMOVE n:label // Removes just the label, not the node!
FOREACH (r IN relationships(path) |
  SET r.marked = true) // Try: MATCH SET RETURN
FOREACH (name IN ["Alice", "Bob"] |
 CREATE (:Person {name: name}))
MERGE (n:Person {name: 'Alice'})
  ON CREATE SET n.created    = timestamp()
  ON MATCH  SET n.accessTime = timestamp(),
      n.counter = coalesce(n.counter, 0) + 1;
CREATE / DROP INDEX ON :Actor(name);
CREATE / DROP CONSTRAINT ON (p:Person)
  ASSERT p.name IS UNIQUE
```

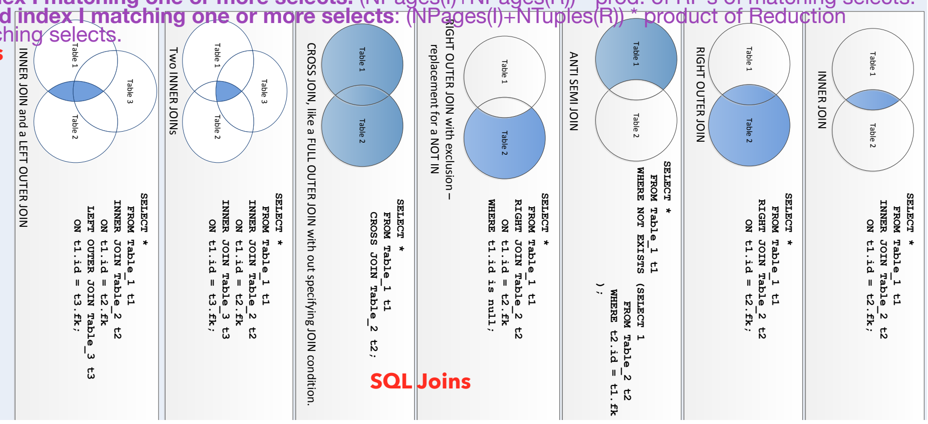
DBMS Security
Objectives of DBMS Security:
secrecy, integrity, availability.
Bell-LaPadula Model: Top Secret (TS) > Secret (S) > Confidential (C) > Unclassified (U)
- Subject S can READ object O iff class(S)≥class(O) - Subject S can WRTIE object O iff class(S)≤class(O)

Approaches to DBMS Security: (1) discretionary and (2) mandatory access control.

(1) in SQL:
```
GRANT select, insert(col), delete,
references(frnKey) ON table TO role WITH
GRANT OPTION; -- Only Owner can CREATE,
ALTER and DROP. Also: CREATE ROLE
role_name; GRANT role_name TO user_name;
REVOKE privilege ON table FROM role
CASCADE; -- if REVOKing select ON a view
FROM its owner, the view gets dropped.
```



Cost Estimates For Single-Relation Plans
- **Sequential scan of file**: NPages(R)
- **Index I on primary key matches selection**: Cost is Height(I)+1 for a B+ tree, about 1.2 for hash index.
- **Clustered index I matching one or more selects:** (NPages(I)+NPages(R)) * prod. of RF's of matching selects.
- **Non-clustered index I matching one or more selects**: (NPages(I)+NTuples(R)) * product of Reduction Factors of matching selects.



SQL Joins

Neo4J Built-In Roles

Privileges	Reader	Publisher	Architect	Admin
Change own password	•	•	•	•
Read data	•	•	•	•
Terminate own query	•	•	•	•
Write/update/delete data		•	•	•
Manage index/constraints			•	•
Terminate others' queries				•