Principles

The relational model

- SQL tables are mathematical relations
- Rows are tuples
- Columns (attributes) are elements in tuples and each has a domain
- The contents of a table determines which tuples are in the relation (the relation state)
    - r(R) = the relation state of a relation R, R is sometimes called the relation schema
- t[X] = A tuple to projected down to only include the attributes in a set X
- **Let SK be attributes of R. SK is a superkey of R iff for every pair of tuples t1, t2 from every legal relation state of R t1[SK] = t2[SK] ⇒ t1 = t2** or equivalently t1 ≠ t2 ⇒ t1[SK] ≠ t2[SK]
    - *That is that deduplicating the set based on checking only a superkey will not lose any information*
- **A superkey of R is a candidate key of R iff it is a minimal superkey iff no proper subset of it is a superkey** iff there does not exist an attribute in SK which when removed gives a superkey
- An unqualified reference to "a key" is usually a candidate key
- SQL uses multisets (bags) instead of sets to enable duplicate rows in result tables but theoreticians use sets

Constraints

- Domain: Limitations of the data types of each attribute are respected
- **Entity integrity: Non-nullable attributes are not null and <u>the primary key is non-nullable</u>**
- **Key: Primary key is a candidate key**
    - An unqualified reference to "the key" means the primary key
    - *Candidate keys other than the primary key can still be captured in a SQL schema using UNIQUE (with NOT NULLs if desired)*
- **Referential integrity: If R1 has a foreign key FK referencing the primary key PK of R2, then ∀ t1 ∈ r(R1). ∃ t2 ∈ R2: (t1[FK] = t2[PK] <u>or t1[FK] is NULL</u>)**
    - Options for dealing with a violation of referential integrity:
        - Do nothing and error — the default
        - Delete referenced rows if deleting, update references with new value if updating — [ON [DELETE | UPDATE] CASCADE
        - Rewrite references as null — SET NULL
        - Rewrite references as the default value for their column — SET DEFAULT
- Semantic attribute integrity: e.g. CHECK in SQL, not really part of the formal model
    - A catch-all for what can't be represented by other constraints
    - Allows richer domain constraints to be captured

SQL

WHERE: Logical operators

- *val BETWEEN low AND high*
- **x LIKE y, patterns:**
    - *[ac…] means match any one character in the list a, c, …*
    - *[^ac…] means match any one character except those in the list a, c, …*
    - **% = zero or more characters (any string)**
    - **_ = exactly one character (any character)**

WHERE: Null handling

- **3-value logic: Comparing a null value using the standard operators gives neither TRUE nor FALSE but instead UNKNOWN — WHERE treats FALSE and UNKNOWN as the same**
- *The SQL standard says due to this NULLs can't violate a UNIQUE constraint, but DBMS compliance varies*
- TRUE AND UNKNOWN = UNKNOWN
- FALSE AND UNKNOWN = FALSE
- UNKNOWN AND UNKNOWN — UNKNOWN
- TRUE OR UNKNOWN = TRUE
- FALSE OR UNKNOWN = UNKNOWN
- UNKNOWN OR UNKNOWN = UNKNOWN
- NOT UNKNOWN = UNKNOWN
- **The usual identity remains the identity and the negation of the usual identity still preserves itself**
  - There is no way of getting away from UNKNOWN without a TRUE or FALSE
- Generally we want to include an **IS NULL** or an **IS NOT NULL** in any boolean expression that might contain nulls in order to specify what to do instead of using this behaviour

DELETE

- DELETE FROM … WHERE … marks all rows that satisfy the WHERE <u>then</u> deletes all the marked tuples
- Note we can only delete rows in this manner, to delete columns we would need to ALTER TABLE

VIEWs

- A VIEW is a relation derived from tables or other views instead of having its own set of values
- Allow frequently used (sub)queries to be abstracted away
- Automatically update when the underlying data changes
- *Views allow us to have an ergonomic logical schema with data duplication etc for easy query writing while having the physical schema that normalisation tells us is optimal*
- Can be virtual (not saved to secondary storage) or materialized (physical) (saved to secondary storage)
- Query results are (virtual) views
- **Query modification (virtual view) — rewrites queries on views in terms of the actual tables** — inefficient (have to recompute data for every use) but quick to initialize (doesn't compute any data until the query optimizer has an idea what we need) — good for simple queries
- **View materialisation (materialized view) — compute and store the entire view for use when queries use it** — more efficient (can compute once then use repeatedly) but more overhead (we are computing the whole table then running the query on it) — good for complex queries

Subqueries

- **A subquery (nested query) is a parenthesized SELECT** that can be used in place of a table or a value
- If a subquery is used where a value is expected and it does not produce exactly one value, a run-time error will occur
  - Such queries should only be written where returning no more than one is guaranteed by the constraints
- **An ordinary subqueries is one that only needs to be evaluated once**
- **A correlated subquery uses attributes from the outer query and hence must be re-evaluated for every tuple of the outer query**

Sets

- Typically used with subqueries
- **SELECT DISTINCT … converts the result from a bag to a set**
    - Distinct a keyword not a function — applies to the whole of each row not just one column, even if you add parentheses (if you only want some columns to not contain duplicates you want to GROUP BY those columns)
- **x op ANY y means: there exists a value in y such that x op y is True**
- **x op ALL y means: for all values in y, x op y is True**
- **x IN y verifies that the tuple x is a member of the set of tuples y** (error if the columns don't match up) or equivalently the value x is a member of the tuple y (and all attributes in y have the same type as x)
    - *Syntactic sugar for = ANY*
- **EXISTS x means x is not the empty set**
    - Note this means EXISTS ((NULL)) gives True
- **We can UNION, INTERSECT, EXCEPT (set difference) subqueries provided the columns match up — these coerce set semantics** (implicitly rewrites each side as SELECT DISTINCT and applies DISTINCT to the result of the operation)
    - **If we really do want bag semantics we write UNION ALL etc.** to no longer place any of the implicit DISTINCTs
    - A UNION ALL B means add all the tuples from the result bag of B to the result bag of A
    - A INTERSECT ALL B means pair up identical tuples in A and B and only return those that managed to pair
    - A EXCEPT ALL B means pair up means pair up identical tuples in A and B and only return those that did not manage to pair and are from A

JOINs

- **SELECT … FROM x, y, … is syntactic sugar for SELECT … FROM X JOIN y JOIN …**
- **JOIN is CROSS JOIN (cartesian product)**
- **NATURAL JOIN is CROSS JOIN filtered down to rows where columns with the same name and type have the same value and these columns projected down to only one occurrence of each**
- JOIN … ON … specifies a condition to be used to filter down the (CROSS) JOIN (for an INNER JOIN this is equivalent to JOIN … WHERE … but this is not the case for OUTER joins as they permit an ON to be falsified but not a WHERE)
- Note can use either NATURAL … JOIN or JOIN … ON, can't have NATURAL … JOIN ON …
- An INNER JOIN (default) only keeps tuples that matched the ON condition
- **A LEFT OUTER join keeps tuples from the left that didn't join with any tuples on the right (as they never matched the ON predicate) and includes them in the result by putting null in the columns that come from B**
  - **Analogously there is a RIGHT OUTER join and [FULL] OUTER join (syntactic sugar for LEFT OUTER JOIN UNION RIGHT OUTER JOIN)**
- *A SEMIJOIN B ON θ gives all tuples in A that matched the condition*
  - *Not in SQL standard but can rewrites as SELECT * FROM A WHERE EXISTS (SELECT 1 FROM B WHERE θ)*
- *A ANTIJOIN B ON … gives all tuples in A that did not match the condition*
  - *Not in SQL standard but can rewrite as SELECT * FROM A WHERE NOT EXISTS (SELECT 1 FROM B WHERE θ)*

Aggregations

- **GROUP by splits the result into groups to which aggregations can be applied — the attributes being grouped on are the only ones that can appear without being inside an aggregation function** as they are the only ones that are guaranteed to be the same across the group and the group is being collapsed down into a single row
- **WHERE … GROUP BY … HAVING …**
    - HAVING applies filtering to the generated results (and so must be used for attributes being grouped over), WHERE applies filtering as the result is being created (so cannot be used for attributes being grouped over)
- **A attribute that is all nulls is skipped over in COUNT** (or SUM, AVG, MIN etc) — this is how COUNT(1) (or COUNT(*)) and COUNT(attribute) can differ

Formal models

Basic Relational Algebra (RA)

- **Projection: $\pi_{a1, ...}$(R) is R with the attributes other than a1, ... dropped** (SELECT a1, … FROM R)
  - **As <u>RA uses set semantics</u>**, if we aren't including the primary key we will lose the extra tuples for any duplicates that arize
- **Selection: σ_c(R) is R filtered down to the tuples that satisfy the condition c** (SELECT * FROM R WHERE C)
  - <u>Note selection is not like SELECT in SQL (projection) but rather WHERE</u>
  - Note nesting is equivalent to conjuncting the conditions and so σ is commutative
- **Rename:** ρ_X(R) renames relation R to X whereas **$\rho_{(b1, …, bn)}$(R) renames the attributes a1, …, an of R to b_1 …, b_n** (SELECT a1 AS b1, …, an AS bn FROM R)
  - Note to rename only some attributes must rename all and just relist of some of the current names
  - Can do both types of rename at once $\rho_{X(b1, …, bn)}$(R)
- Note <u>p</u>i for <u>p</u>rojection, <u>s</u>igma for selection, <u>r</u>ho for <u>r</u>ename
- Can use set operations (∪, ∩, −) provided the relations are type compatible (have the same number of attributes and for each attribute position the attributes in that position have the same domain)

Basic Relational Algebra (RA): Joins

- **Product join: R × S** takes the cartesian product of R and S (R CROSS JOIN S)
- **Natural join: R ∗ S is** R × S filtered down to rows where columns with the same name and type have the same value and these columns projected down to only one occurrence of each
    - *Is often also denoted R ⋈ S*
    - If columns mean the same thing but are named differently (or named the same but mean different things), use ρ first
    - **If there are no matching columns, a natural join is a cross join (this is also the case in SQL)**
- **(Theta) join: $R \bowtie_\theta S$ is syntactic sugar for $\sigma_\theta(R \times S)$ (R JOIN S ON θ)**
    - Being precise, it's only a theta join if the condition is a ◦ b and ◦ ∈ {≤, ≥, <. >, =, ≠} — if ◦ is =, it is specifically an equijoin

Extended relational algebras

- Let R(Z), S(X) be relation states with relation schemas Z and X. Then, R(Z) ÷ S(X) = {t: $\forall t_S \in S.\ \exists t_R \in R$: t = $t_R[Z - X]$ and $t_R[X] = t_s$} where Y = Z − X or equivalently **$R(Z) \div S(X) = \pi_{Z-X}(R) - \pi_{Z-X}((S \times \pi_{Z-X}(R)) - R)$**
  - We define $t_A[B]$ to be the tuple t_A projected down to only the columns it shares with relation schema B
  - We can read the first definition as: The set of tuples t from R projected onto Z − X such that t combines with every row in S to produce a tuple in R
- **a1, … ɣ fa aa fb ab … (R) is SELECT a1, …, fa(aa), fb(ab), … FROM R GROUP BY a1, …**
- *A $\ltimes_\theta$ B is A SEMIJOIN B ON θ — can rewrite as $\pi_{a1, \ldots, an}(R \bowtie_\theta S)$ where a1, …, an are the attributes of R*
- *A $\triangleright_\theta$ B is A ANTIJOIN B ON θ — can rewrite as $R - \pi_{a1, \ldots, an}(R \bowtie_\theta S)$ where a1, …, an are the attributes of R*
- Pure relational algebra lives in idyllic set-based theory land where null does not exist and so doesn't support any type of outer join, but some people just can't help themselves:
  - A ⟕$_\theta$ B is A LEFT OUTER JOIN B ON θ
  - A ⟖$_\theta$ B is A RIGHT OUTER JOIN B ON θ
  - A ⟗$_\theta$ B is A [FULL] OUTER JOIN B ON θ

Tuple relational calculus

- Relational algebra is a procedural-esque model using operators whereas relational calculus is a declarative-esque model using set builder with first-order-logic
- Codd's Theorem: Relational algebra and relational calculus are equivalent in expressive power
  - Corollary: As relational algebra is equivalent in expressive power to (pure) SQL, so is relational calculus
- The relation schema defines a predicate and the relation state defines the set of propositions (tuples can be viewed as instatiantions as the predicate of the schema) that are true
  - Closed world assumption: Every true proposition from that predicate is in the relation state and so everything not in the relation state is false
- SELECT Fname, Lname FROM Employee WHERE Salary < 30000 translates to {t.Fname, t.Lname | Employee(t) ∧ Salary < 30000}
  - LHS of | is our projection
  - RHS of | is range relations (defining which relations the tuple variables are ranging over), selections, and joins
- **An expression (set) is called safe iff it is guaranteed to contain a finite number of tuples** e.g. {t | EMPLOYEE(t)} is safe but {t | NOT(EMPLOYEE(t))} is unsafe
  - Every <u>safe</u> expression can be rewritten as relational algebra

Tuple relational calculus: Logic

- **If we want p ⇒ q we have to write it as ¬p ∨ q**
- Can use existential quantifier to bind variables but domain is still specified by range relation
    - SELECT Fname, Lname, Address FROM Employee e WHERE EXISTS (SELECT 1 FROM department d WHERE Name = “Research” AND d.Dnumber = e.Dno) translates to {t.Fname, t.Lname, t.Address | EMPLOYEE(t) ∧ (∃d)(DEPARTMENT(d) ∧ d.Dname = 'Research' ∧ d.Dnumber = t.Dno)} where t is a free variable and d is a bound variable
- **If we want ∀x ∈ Y. p, we have to write it as (¬∃x)(Y(x) ∧ ¬p)**

Domain relational calculus

- **Variables are attributes of tuples instead of tuples**
- The following are atomic formulas
  - **$R(x_1, \ldots, x_k)$ is a range relation**, asserts that $x_1, \ldots, x_k$ are the attributes of the relation R — commas and spaces may be omitted if it does not hurt clarity
  - Theta conditions
- The following are compound formulas where F, F_1, F_2 are compound or atomic formulae:
  - NOT(F)
  - F_1 OR F_2
  - F_1 AND F_2
  - $(\forall x)(F)$
  - $(\exists x)(F)$
- E.g. SELECT Bdate, Address FROM EMPLOYEE WHERE Fname = ‘John’ AND Minit = ‘B’ AND Lname = ‘Smith’ is written as $\{u, v \mid (\exists q)(\exists r)(\exists s)(\exists t)(\exists w)(\exists x)(\exists y)(\exists z)$ $(\text{EMPLOYEE}(qrstuvwxyz) \text{ AND } q = \text{'John' AND } r=\text{'B' AND } s=\text{'Smith'})\}$

Normalisation

Functional dependencies

- **A functional dependency X → Y is an assertion that the values of the attributes of a tuple that are in the set Y are completely determined by the values of the attributes that are in the set X**
  - A **functional dependency X → Y means the relation acts like a (partial) function** from the set of assignments for X to the set of assignments for Y (no valuation of X has more than one distinct valuation of Y associated with it)
  - **Formally, X → Y iff for all pairs of tuples t1, t2 in all possible relation states t1[X] = t2[Y] ⇒ t1Y] = t2[Y]**
  - **Come from real world constraints**, looking at a particular relation state we can only verify whether a certain functional dependency does not exist not whether it does exist
- If K is a superkey of a relation with schema R, then ∀S ⊆ R. K → S
- If X → R and R is the relation schema that X belongs to, then X is a superkey

Implied functional dependencies

- **Let F be the set of functional dependencies of R. F^+ = set of all implied functional dependencies from F**
- **Armstrong's inference rules (minimal sound and complete rules for deriving F^+):**
  - **AB denotes A U B**
  - **Reflexive: $Y \subseteq X \Rightarrow X \rightarrow Y$**
  - **Augmentation: $X \rightarrow Y \Rightarrow XZ \rightarrow YZ$ for all Z**
  - **Transitive: $X \rightarrow Y$ and $Y \rightarrow Z \Rightarrow X \rightarrow Z$**
- Useful rules that follow from Armstrongs':
  - Decomposition: $X \rightarrow YZ \Rightarrow (X \rightarrow Y$ and $X \rightarrow Z)$ [e.g. $YZ \rightarrow Y$ by reflexive, so $X \rightarrow Y$ by transitive]
  - Union: $(X \rightarrow Y$ and $X \rightarrow Z) \Rightarrow X \rightarrow YZ$ [$XZ \rightarrow YZ$ and $XX = X \rightarrow XZ$ by augmentation, so $X \rightarrow YZ$ by transitive]
  - Composition: $(X \rightarrow Y$ and $A \rightarrow B) \Rightarrow XA \rightarrow YB$ [similar proof to union]
  - Pseudo transitivity: $(X \rightarrow Y$ and $YZ \rightarrow W) \Rightarrow XZ \rightarrow W$ [similar proof to union]

Properties of a good decomposition

- Each tuple should only represent one entity to make schema easier to understand
- Redundancy should be minimized to minimize storage requirement and minimize the risk of anomalies being introduced
  - Update anomaly = A single logical update requires updating several tuples so could accidentally only update some
  - Insert anomaly = An insert cannot occur as data is not known for non-nullable fields
  - Delete anomaly = A delete cannot occur as data we don't want to lose is in the tuple containing the data we want to delete
- Use of nulls should be minimized to minimize storage requirement and make queries easier to understand
- **Joins that will need to be performed should be on keys**
- **Lossless decomposition (non-additive join property)**: Natural join of decomposition gives exactly the original table for any legal relation state — no spurious tuples or lost tuples
  - **Suppose we have decomposed R into R_1, R_2 and we had set of functional dependencies F on R. R_1 and R_2 is a lossless decomposition iff $(R_1 \cap R_2 \rightarrow R_1 \setminus R_2) \in F^+$ or $(R_1 \cap R_2 \rightarrow R_2 \setminus R_1) \in F^+$**
- **Dependency preservation property: Suppose we have decomposed R into $R_1, \ldots, R_n$ and we had functional dependencies F on R. We have dependency preservation iff ((F holds on each of $R_1, \ldots, R_n$) $\Rightarrow$ F holds on R)**
  - If we don't have this, we have to do joins to verify every modification which is inefficient
- We cannot always get lossless and dependency preserving at the same time, in such cases we should opt for lossless

3NF

- When we say a key in the context of normalisation we mean a candidate key
- ***Prime attribute = an attribute that occurs in a key***
- ***Non-prime attribute = an attribute that does not occur in any key***
- **First normal form (1NF): Attributes are atomic** (not multi-valued (really correspond to multiple rows) or composite (really correspond to multiple columns)) **and there exists a key** (e.g. no duplicate rows)
- **Second normal form (2NF): Every <u>non-prime</u> attribute is** <u>fully</u> dependent on every key (that is dependent on every key (this is the definition of a key) <u>and **not dependent on any proper subset of any key**</u>)
- **Third normal form (3NF):** No transitive functional dependency from a key to a <u>non-prime</u> attribute (that is **no functional dependencies between non-prime attributes**)
- <u>**Each normal form has the condition that the table is already in the previous normal form**</u>

BCNF (Boyce-Codd normal form) (“3.5”NF)

- **BCNF: Every functional dependency has LHS (possibly a superset of) a key or is trivial (RHS is subset of LHS)**
  - 3NF allows functional dependencies X → Y iff Y\X is a set of prime attributes or X is a super key or Y is a subset of X but BCNF only allows the latter two cases
  - <u>**In BCNF we are no longer restricting ourselves to checking only functional dependencies to non-prime attributes**</u>
- In BCNF, no redundancies <u>based on functional dependencies</u> remains
- *Any relation in 1NF can be turned into 3NF but may not be able to be turned into BCNF without breaking dependency preservation*
  - *R(x, y, z) with functional dependencies {x, y} → {z} and {z} → {y} is trivially in 3NF as there are no non-prime attributes but is not in BCNF due to the {z} —> {y} fd. Decomposing this into R_1(z, y) and R_2(z, x) causes each relation to be in BCNF but this is not a dependency preserving decomposition and so best practice would be to leave it in 3NF.*
- If there is at most one overlapping key, a table in 3NF is necessarily in BCNF

“So help me Codd”

- “I do hereby solemnly wear that each non-prime attribute is dependent on the key (1NF), the whole key (2NF), and nothing but the key (3NF), so help me Codd”
- Really we should say “I do hereby solemnly swear that each non-prime attribute is dependent on every key (1NF), the whole of every key (2NF), and nothing but every key (3NF), so help me Codd” but that kind of ruins the reference
- If we remove the non-prime qualification we get a description of BCNF — “I do hereby solemnly swear that each attribute is dependent on every key (1NF), the whole of every key (2NF), and nothing but the whole of every key (BCNF), so help me Codd”

Security and access control

SQL injection and prepared statements

- **SQL injection:**
  - **Add ORs**
  - **Add UNIONs**
  - **Add semicolons**
  - **Use comments (--)**
- When we create a prepared statement the parameterized query is sent to the database to be parsed and have its query plan created
  - When we execute the query the already prepared query plan is carried out with the provided data
  - More efficient and has data and instructions separated for free (and effective) SQL injection prevention
- Could use input sanitization but prepared statements are just better
- Security by obscurity should not be relied on for anything but could form part of a defense in depth approach:
  - Treat the schema as sensitive information and don’t use common table/attribute names in it
  - Don’t allow the SQL server error messages to be shown to the end-user and make the errors you generate instead as vague as possible

Discretionary access control

- **GRANT or REVOKE privileges to run each command [SELECT | UPDATE | …] ON a relation TO a user**
  - *Is implemented as an access matrix with rows of subjects and columns of objects and cells of privileges*
    - *Access matrix is often but not always implemented as a relation*
- **If we only want to allow a user to SELECT certain columns of a relation we can CREATE a suitable VIEW and GRANT SELECT ON that view TO them instead**
- **Appending WITH GRANT OPTION to the end of a GRANT allows the user to grant privileges to other users** — if privileges are revoked from the granting user, they will also be removed from the users they granted it to (unless the same privileges were also granted by another user who still has those privileges)
- Role based access control works like group level permissions for files:
  - **Can make GRANTs TO roles instead of users**
  - **CREATE/DESTROY ROLE rolename**
  - **GRANT ROLE rolename TO username**

Mandatory access control

- Each subject (e.g. user) has a security level (clearance)
- Each object (e.g. relation, attribute) has a security level (classification)
- **Simple security restriction: No read up**
- **Confinement restriction** (flow control)**: No write down** — in practice no write up either as seeing whether or not constraints prevented your change being applied is an unavoidable form of reading
- **Classification of a tuple is the highest classification of its attributes**
- **A relation that would otherwise have schema R(A_1, …, A_n) has schema R(A_1, C_1, …, A_n, C_n, TC) where C_1, …, C_n are the respective classifications of attributes A_1, …, A_n and TC is the tuple classification**
- **Users can only add/modify tuples with TC equal to their clearance**
- **Users can only read tuples with TC less than or equal to their clearance**
- **We can create copies of a tuple with some columns NULLified to allow users with lower classifications to see some of it:**
  - **Apparent key = what the key of R(A_a …, A_z) would be where A_1, …, A_z are the non-null attributes**
    - If this key is composite, all attributes should have the same classification
    - There should not be any attributes with lower classification than it
  - **Polyinstantiation = creating multiple tuples with the same apparent key** to provide different tuple classifications
    - *This can happen systematically or by the natural actions of users of different levels*

Statistical databases

- Statistical databases have sensitive rows but insensitive aggregations over suitably general queries
- **Approaches for deciding whether the population selected by a query is suitably general:**
  - **Require population to be above a certain size**
  - **Limit the rate at which similar queries can be made by the same user**
  - **Have a set list of populations that are allowed and only allow people to use these and their unions**
  - Introduce noise to the responses — ideally noise would be added to the stored data as if it is regenerated for each query it could be averaged out by bad actors
- **Randomised response: Introduce noise** (see above) **to give plausible deniability**
  - E.g. if recording answer to a sensitive (in the affirmative) binary question, record truthful answer in DB with 50% probability and record YES with 50% probability
    - When getting number of YESs from table, subtract number of NOs in table from it to get the true figure
    - When getting number of NOs from table, double it to get the true figure

Flow control

- Each memory region is assigned a security class
  - No read up
  - No write down
- Very hard to stop all implicit/indirect flows