4DB3 Review

Relational Model

Logical Model – the conceptual schema which shows the name of the relationship and the attributes of the tuple.

Physical Model – the way relations are stored, describes the files and indexes uses.

Constraints

Integrity constraint is a property that must be satisfied by all meaningful database instances. Tuple constraint expresses conditions on the values of each tuple, domain constraint is a tuple constraint that involves a single attribute (i.e GPA <= 4 AND GPA >= 0)

Keys

Key is a set of attributes that uniquely identifies tuples in a relation. A superkey is a superset of a key (i.e. a set of attributes which are unique together). A candidate key is a minimal superkey (no subset superkey exists in the minimal superkey).

Primary keys are underlined, no nulls allowed in the attributes of the primary key. Foreign keys require that the values on a set X of attributes of relation R must appear as values for the primary key of another relation (i.e. attributes used from another relation must be a primary key in the relation it is taken from)

Referential Integrity

If all foreign key constraints are enforced, referential integrity is achieved. Non-existent FK's should be rejected when tried to insert. You can reject deletion of referred tuples in the original table, allow cascading deletion, default or set to null upon deletion of referred tuple.

Relational Operations

Selection on predicate, Cross product (pair all tuples of one relation, with all tuples of another relation), Union (list tuples from both relations), Difference (remove all tuples from the first relation that are located in the second relation), Intersect (only include tuples that are in both relations), Natural Join (connect relations on equal attribute names, merge pairs of tuples).

Entity Relationship Model

Allows us to sketch database schema designs. Entities are a thing or object, Entity sets are a collection of similar entities (i.e. Beers), Attributes are properties of an entity (i.e. integers or characters).

Entities are rectangles, Attributes are ovals, Relationships (form the tables into useful data) connect two or more entity sets (represented by a diamond) (i.e. Sells).

Different Types of Relationships

Many-Many (entity of either set can be connected to many entities of the other set i.e. bar sells many beers, beer is sold by many bars).

Many-One (entity of the first set is connected to at most one entity of the second set, but entity of second set can be connected to zero, one, or many entities of the first set i.e. a drinker has one fav beer but a beer can be many drinkers' fav). Total vs. Partial (if every entity of the first set has to be connected to one of the second set its Total).

One-One (each entity of either entity set is related to at most one entity of the other set i.e. Best-seller)

Rounded Open Arrow for “exactly one”, each entity of the first set is related to exactly one entity of the target set (if the arrow is closed, its partial)
Underline key attributes and label roles (if they appear more than once in a relationship i.e. Husband and Wife on Drinkers) on edges of relationship to entity set arrows.

Weak Entity Sets

An entity set is said to be weak if in order to identify entities of E uniquely, we need to follow one or more many-one relationships from E and include the key of the related entities from the connected entity sets. Double rectangle means weak entity set, double diamond means supporting many-one relationship.

ISA Hierarchies (‘is a’)

SQL

DDL (Data Definition Language) & DML (Data Manipulation Language)

SQL is a query language for getting info from a db (DML), however it has a DDL component for describing schemas of a db.

SELECT – FROM – WHERE (PREDICATE)

Comparing a value to a NULL yields an UNKNOWN.

Subqueries can only be used as a value if it is guaranteed to return one tuple.
Ex.
SELECT bar FROM Sells WHERE beer = ‘Miller’ AND price = (SELECT price FROM Sells WHERE bar = ‘Joe’’s Bar’ AND beer = ‘Bud’)
This returns the bars, where Miller is sold at a price equal to the price of Bud at Joe’s bar.

ANY Operator

X = ANY(subquery) is a Boolean condition, tuples must have ONE component, compares to any tuple in the subquery.

ALL Operator

Only true if true for all tuples of subquery.

IN Operator

IN operator is only true when member of relation is found in subquery. In only loops over once, while Join loops multiple times.
Ex.
SELECT * FROM Cartoons WHERE LastName IN (‘Jetsons’,’Smurfs’,’Flintstones’)

EXISTS Operator

True if and only if the subquery result is NOT empty.

UNION, INTERSECT, AND DIFFERENCE

Subquery UNION Subquery

Bag Semantics vs Set Semantics

SELECT-FROM-WHERE uses bag semantics (duplicates exist), where as UNION, INTERSECTION and DIFFERENCE operations apply set semantics (duplicates are eliminated).

DISTINCT

Force a result to be a set using SELECT DISTINCT, force a result to be a bag by using ALL (ex. UNION ALL)

<Relation> NATURAL JOIN <Relation 2>, <Relation> CROSS JOIN <Relation 2>, <relation> JOIN <relation> ON <condition>

Create Table

CREATE TABLE <name> (<attribute name> <type>);

DROP TABLE <name>;

Declaring Keys

Can list an attribute as UNIQUE or PRIMARY KEY. There can only be one PK for a relation, but several UNIQUE attributes, PK's cannot have a NULL in the tuple, where as UNIQUE can have NULLs.

Ex.

```
CREATE TABLE sells (
Bar CHAR(20),
PRIMARY KEY (Bar)
);
```

Foreign Keys

Ex.

```
CREATE TABLE Sells (
Bar CHAR(20),
Beer CHAR(20),
FOREIGN KEY (beer) REFERENCES Beers(name)
);
```

Enforcing FK Constraints (referential integrity)

Insert or Updates to **a table** where the FK does not exist in the referenced table must be REJECTED.

Deletions or Updates to the **referenced table** can be Defaulted (reject modifications), Cascade (Make the same changes to all tables referencing FK), or Set NULL.

Set Policy –

```
CREATE TABLE Sells (
bar CHAR(20),
beer CHAR(20),
FOREIGN KEY(beer) REFERENCES Beers(name)
ON DELETE SET NULL
ON UPDATE CASCADE);
```

CHECKS

Checked on insert or update only, CHECK (Condition), can apply to an attribute or to the relation-scheme element.

DB Modification

INSERT INTO <relation> VALUES (<list of values>), can also specify attributes:

Eg. INSERT INTO Likes(beer, drinker) VALUES ('Bud', 'Sally'); purpose of doing it this way is to add only what is known into the tuple, leaving the rest NULL.

DEFAULT

Default is declared right after the declaration of an attribute. Addr CHAR(50) DEFAULT '123 Sesame St.'

Can Insert Subqueries

INSERT INTO <relation> (<subquery>);

Deletion

Deletion proceeds in 2 stages, marks all tuples for which the condition is satisfied, then delete the marked tuples, it does not delete immediately after identifying a satisfied condition in the query.

UPDATE

UPDATE <relation> SET <assignment> WHERE <condition on tuple>;

Eg. UPDATE drinkers SET phone = '555-1212' WHERE name = 'Billy';

JOINS

Dangling is when you join two relations but there is missing data based on the joining relation (results in NULLS).

Left-Outer-Join: Nulls in columns from missing relation data in right relation (tuples are only joint if they exist in L).

Eg. L natural left outer join R

Right-Outer-Join: Nulls in columns from missing relation data in left relation (tuples are only joint if they exist in R).

Full-Outer-Join: Combines the tables, leaving nulls where no information is present (all tuples are represented).

Inner Join: Only tuples in both tables are shown, no NULLS are present.

Aggregation

SUM, AVG, COUNT, MIN, MAX can be applied to columns in a SELECT clause.

Eg. SELECT AVG(price) FROM Sells WHERE beer = 'Bud';

Nulls are ignored in aggregation, they never contribute to the sum,avg,etc, if no non-NULL values exist, the results of the aggregation is NULL.

Grouping

SELECT-FROM-WHERE followed by a GROUP BY (attributes), results in aggregations only being applied within each group.

Relational Algebra

Selection

R1 := σ_C (R2)

C is a condition that refers to attributes of R2.

Projection

R1 := π_L(R2)

L is a list of attributes from the schema R2 (this controls the columns of the relation to display).

Extended projection

Allows people to apply arithmetic on the variable L (eg. A + B -> C)

Product

R3 := R1 X R2

Every tuple of relation 1 is paired with every tuple of relation 2.

Theta Join

R3 := R1 $\bowtie_C$ R2

Take the product of R1 x R2 then apply select C, where c is a condition.

Natural Join

$R3 := R1 \bowtie R2$

Equate attributes of the same name, only show one copy of each pair of equated attributes.

Rename

$R1 := \rho_{R1(A1,\ldots,An)}(R2)$

Rename the attributes of relation R2.

Union

R1 U R2

The resulting relation contains all the tuples in both relations (duplicates included)

Intersection

$R1 \cap R2$

The resulting relation contains the minimum between both relations for each tuple (Eg. R1(1,2), R1(1,2) and R2(1,2), then the result is just R3(1,2), the min(2, 1)=1).

Difference

R1 – R2

The resulting relation contains the tuples of the first set, minus every occurrence of a tuple in the second set (1 to 1 removal though).

Relational Algebra Defaults to SET SEMANTICS (no duplicates after operation)

Extended Algebra

Duplicate Elimination

$R1 := \delta(R2)$

Only one copy of each tuple from R2 is in R1.

Sorting

$R1 := \tau_L(R2)$.

L is a list of the attributes of R2, R1 is the resulting list of tuples of R2 sorted first on the value of the first attribute of L, then second, etc.

Aggregation/Grouping

$R1 := \gamma_L(R2)$.

L is a list of the attributes of R2 to group by.

Eg.

$\gamma_{A,B,AVG(C)->X}(R)$

Group by (A,B) first then average within groups by C.

AVG,MAX,MIN,SUM,COUNT

Views and Indexes

View – a view provides a mechanism to hide certain data from the view of certain users, defined in terms of stored tables and other views. Can be **virtual** or **materialized**, virtual being not stored in the database (i.e. just a query to construct relation), while materialized is an actually constructed and stored table.

CREATE [MATERIALIZED] VIEW <name> AS <query>; (default is virtual, generally impossible to modify a virtual view because it doesn't exist)

Insertion and update on view

If trying to insert into a view, you will probably need to translate the insert into multiple insertions across the base tables.

Eg.

INSERT INTO table1 VALUES (value1, value2);

INSERT INTO table2 VALUES (value3, value2);

Just to do a modification of a view.

Indexes

Index is a data structure used to speed access to tuples of a relation, given values of one or more attributes. They speed up searches for a subset of records, based on values in certain("search key") fields.

CREATE INDEX attributeindex ON relation(attribute);

PRO: Index can speed up queries that can use it. **CON:** An index slows down all modifications on its relation because the index must be modified as well.

Clustered vs. Unclustered Index

If the order of index data entries is the same as the order of data records, then called clustered index (can only be clustered on at most ONE search key)

Selecting Indexes

Attributes in the WHERE clause are candidates for index keys (exact match conditions (equality) suggest hash index, a range query suggests tree index). Choose an index that benefits as many queries as possible.

Database Design

Avoid Redundancy, can lead to **update anomalies** (if an attribute is changed, how do we remember to change all the tuples with the same attribute) and **deletion anomalies** (can lose track of data if table is redundant and nothing is tracking specific data that can be separated from tuple).

Functional Dependencies

X->Y is an assertion about relation R, that whenever 2 tuples of R agree on all attributes in X, they also agree on all attributes in Y.

Eg. X->Y, X functionally determines Y

Have an FD only if it holds for every instance of the relation, need to determine knowledge domain.

Eg. Surname doesn't determine country, and neither does city, but it may look like that rule follows, but it's not an FD.

Keys

K is a key for R if K functionally determines all of R, and no subset of K does.

Superkey

K is a superkey for R if K contains a key for R. (superset of a key) K -> R (all attributes of a relation must be included on the right hand side if it is a key/superkey)

Dependency Inference

Even if an FD isn't explicitly stated, we can infer FD's. Eg. If A->B and B->C then A->C.

Closure Test

An easy way to test is to compute the closure of attributes Y denoted by Y+.

Eg.

Test A-> C given A->B, B->C

Base Case: A+ = {A}

Induction: A->B satisfies, A+ = {A,B} B->C satisfies, A+ = {A, B, C}

C in A+ therefore A -> C holds.

Armstrong's Axioms

Reflexivity – if Y is a subset of X then X -> Y

Augmentation – if X -> Y, then XZ -> YZ for any Z

Transitivity – If X->Y and Y ->Z then X -> Z

Union – if X->Y and X->Z then X -> YZ

Decomposition – if X -> YZ then X -> Z and X -> Y

Projection

Given a relation R, the set F of FDs that hold in R, and a relation R_i subset of R, determine the set of all FDs F_i that follow from F and involve only attributes of R_i.

Eg.

ABC with FD's A->B, B->C. Project onto AC.

A+ = ABC; yields A->B, A->C

C+ = C; yields nothing

B+ = BC; yields B->C

Resulting FDs: A->B,A->C and B->C

Project onto AC: A->C

Minimal Cover

Right sides are single attributes, No FD can be removed, no attribute can be removed from a left side.

Normalization

Goal of Decomposition

Eliminate redundancy by decomposing a relation into several relations

Lossless join decomposition

When we join decomposed relations, we should get exactly what we started with.

$r = r1 \bowtie \ldots \bowtie rn$ where $ri = \pi_{Ri}(r)$

Lossy Decomposition

When relations are joined together, lossy decompositions yield MORE tuples than they should when relations are joined, this leads to LESS information (because information is not necessarily true, so it corrupts data).

Testing for Losslessness

Lossless iff either FD(R1 INTERSECT R2) -> R1 is in F+ or FD(R1 INTERSECT R2) -> R2 is in F+.

Basically, if all attributes common to both R1 and R2 functionally determine ALL the attributes in R1 or R2.

If R1 INTERSECT R2 forms a superkey of either X or Y, decomposition of R is a lossless decomposition.

Eg.

Given:

Lending-schema = (branch-name, branch-city, assets, customer-name, loan-number, amount)

Required FD's:

branch-name-> branch-city, assets

loan-number ->amount, branch-name

Decompose Lending-schema into two schemas:

Branch-schema = (branch-name, branch-city, assets)

Loan-info-schema = (branch-name, customer-name, loan-number, amount)

Show that decomposition is Lossless Decomposition

Branch-schema = (branch-name, branch-city, assets)

Loan-info-schema = (branch-name, customer-name, loannumber, amount)

Since Branch-schema ∩ Loan-info-schema = {branch-name}

We are given: branch-name -> branch-city, assets (functionally determines all attributes in Branch Schema)

Thus, this decomposition is Lossless decomposition

BCNF (Boyce-Codd Normal Form)

R is in BCNF whenever X->Y is a nontrivial FD that holds in R, and X is a superkey (Y not contained in X and X is a superset of a key). Basically, the only FDs that hold in BCNF are those that are the result of key(s). This is not dependency preserving.

Eg.

Continue decomposing a relation until the decomposed relations only use the keys they need.

Relation: Drinkers(name, addr, beersLiked, manf, favBeer)

F = name->addr, name -> favBeer, beersLiked->manf

Key = name,beersLiked

 Pick BCNF violation: name->addr.

 Closure: {name}+ = {name, addr, favBeer}.

 Decomposed relations:

 Drinkers1(name, addr, favBeer), only key now is {name}

 Drinkers2(name, beersLiked, manf), however this still uses {name, beersLiked}, so decompose this.

Result:

Drinkers1(name, addr, favBeer)

Drinkers3(beersLiked, manf)

Drinkers4(name, beersLiked)

Dependency Preservation

All the original FDs should be satisfied. BCNF does not satisfy this, projected dependencies are not equivalent to the original FDs. 3NF lets us avoid the problem of decomposing yet breaking the original FDs.

3NF (3rd Normal Form)

An attribute is **prime** if it is a member of any key.

X->A violates 3NF iff X is not a superkey AND also A is not a prime.

Eg.
R = ABC, FDs = AB->C and C->B
Keys are AB and AC.
Therefore A,B,C are prime.
C -> B violates BCNF (because C is not a superkey), but C and B are primes so it does not violate 3NF.
However 3NF results in anomalies.
3NF vs. BCNF
3NF preserves dependencies and lossless join, but results in anomalies, while BCNF results in dependency lost, but results in NO anomalies and keeps lossless join.

Transactions

Transaction is the DMS's abstract view of a user program: sequence of reads and writes.
Concurrency is achieved by the DBMS which interleaves actions (reads/writes of DB objects) of various transactions.
Transaction Properties (ACID)
Atomicity – transaction is either performed in its entirety or not performed at all (allows things to be undone).
Consistency – transaction must take the database from one consistent state to another.
Isolation – Transaction should appear as though it is being executed in isolation from other transactions
Durability – Changes applied to the database by a committed transaction must PERSIST, even if a system fails before all changes are reflected on the disk.
Schedules
Eg.
S: R1(A),R2(A),W1(A),W2(A),A1,C2; (where A1 is abort 1, C2 is commit 2) and the numbers designate the two transactions.
Serial schedule
Schedule that does not interleave the actions of different transactions.
Equivalent schedule
The effect on the set of objects in the db of executing the first schedule is identical to the effect of executing the second schedule
Serializable schedule
A schedule that is equivalent to some serial execution of the transaction (all operations of one transaction occur before the other transaction).
Conflict
Two operations in a schedule are in conflict if they belong to different transactions, access the same item A, and one of the operations is a write(A).
What can go wrong:
Reading uncommitted data (WR) aka "dirty reads"
Unrepeatable reads (RW) (reading data that is no longer recent)
Lost updates (WW) (overwriting basically)

Recoverable Schedules and Avoid Cascading Aborts

A recoverable schedule only commits after the transaction it depends on commits or aborts. (not necessarily ACA).

Avoid cascading aborts by reading in a transaction only after the first transaction has committed or aborted (prevents requiring both transactions to be aborted).

Precedence Graph Test

To test if a schedule is conflict-serializable you build a graph of all transactions. Edge from Ti to Tj if it makes an action that conflicts with one of the Tj, if there are no cycles, its conflict serializable.

Eg.

R2(A); R1(B); W2(A); R3(A); W1(B); W3(A); R2(B); W2(B);

1 ->(B) -> 2 ->(A) -> 3, conflict serializable.

Eg2.

R2(A); R1(B); W2(A); R2(B); R3(A); W1(B); W3(A); W2(B)

1 -> (B) -> 2 -> (A) -> 3

2 -> (B) -> 1, cycle, not conflict serializable

Strict

A schedule is strict if a value written by a transaction is not read or overwritten by another transaction until the original transaction aborts or commits.

Locking

L_i(A) – transaction i acquires lock on element A

U_i(A) – transaction i releases lock on element A

Types of Locks

Shared lock (for reading) S(A), Exclusive lock (for writing and reading) X(A)

Strict 2PL (two phase locking)

Two rules:

1. Each transaction must obtain an S lock on object before reading, and an X lock before writing.
2. All locks held by a transaction are **released only when the transaction COMPLETES**

If a transaction holds an exclusive lock on an object, no other transaction can get a lock S or X on that object.

Strict 2PL allows only schedules whose precedence graph are acyclic (i.e. serializable, the transactions don't depend on one another, they can run completely isolated).

The locking protocol only allows safe interleaving of transactions. This can result in deadlocks (both transactions waiting for the other to release a resource before completing).

2PL (two phase locking)

Variant of Strict 2PL

Relaxes the second rule, so that transactions can release locks before the end (commit/abort)

*The added rule of a transaction not allowed to request additional locks once it releases any lock.

Each transaction is executed in 2 phases, growing phase(obtaining) and shrinking (releasing)

Performance of Locking

Blocking, Aborting or Deadlocking transactions takes performance penalties.

Good heuristic – if more than 30% of transactions are blocked, reduce the number of concurrent transactions.

Locking overhead is primarily due to dleays from blocking.

Lock the smallest sized object

Reduce likelihood of 2 transactions needing the same lock.

Reduce time transactions hold locks

Reduce hot spots (object that is frequently accessed, causing blocking delays)

Locking Granularity

Size of data items to lock.

Coarse granularity implies, few locks, little locking overhead, but large chunks of data so high chance of conflict, concurrency may be low.

Fine granularity implies many locks, high locking overhead, locking conflict is less likely.

Intention locks

Used when S and X locks can't detect if an object is locked based on hierarchy (ie. A file might not be locked, but the relation in the file is, then the file being accessed does not prevent the relation from being changed).

Solution: Set intention locks on coarse grained data that contains the fine-grained data to be locked.

3 types of intention locks:

Intention-shared (IS) – indicates that a shared lock will be requested on descendant nodes

Intention-exclusive (IX) – indicates that an exclusive lock will be requested on descendant nodes

Shared-intention-exclusive (SIX) – indicates that the current node is locked in a shared mode but an exclusive lock will be requested on some descendant nodes.

Deadlocks

Cycle of transactions waiting for locks to be released by each other.

Detection

Detect deadlocks automatically, abort a deadlocked transaction (the victim). Timeout based detection, but may abort unnecessarily (pessimistic).

Create waits-for-graph

Nodes are transactions, edge if transaction is waiting for another transaction to release a lock. Edge is added when lock request is queued, removed when request is granted. Deadlock resolved by aborting a transaction in the cycle, releasing its locks (different criteria can be used: fewest/most locks, farthest from completion, etc.)

Deadlock Prevention

Prevent deadlocks by giving each transaction a priority, assign based on timestamps, **lower timestamp** indicates **higher priority**.

Two possible policies (assume Ti wants lock that Tj has):
Wait-Die – if Ti has higher priority, Ti waits for Tj otherwise it aborts.
Wound-Wait – If Ti has higher priority, Tj aborts; otherwise Ti waits.