CS336 Final Guide

Relational Algebra

Projection

<table>
<tr><td>Purpose</td><td>Selects all tuples from the specified fields (removing duplicates)</td></tr>
<tr><td>Syntax</td><td>π _[list of fields]_ (R)</td></tr>
<tr><td>SQL Equivalent</td><td>select ___, ____, ____</td></tr>
<tr><td>Example</td><td>$\pi_{name,gpa}$ (students) =
<table><tr><td>name</td><td>gpa</td></tr><tr><td>Jones</td><td>3.4</td></tr><tr><td>Smith</td><td>3.2</td></tr></table></td></tr>
</table>

Selection

<table>
<tr><td>Purpose</td><td>Sets a condition of what is to be selected from a relation</td></tr>
<tr><td>Syntax</td><td>σ _[condition]_ (R)</td></tr>
<tr><td>SQL Equivalent</td><td>where _[condition]_
Condition can be any of the following:
= equals
< > not equals
< less than
> greater than</td></tr>
<tr><td>Example</td><td>$\pi_{name}(\sigma_{age=18}(\text{students}))$ =
<table><tr><td>name</td></tr><tr><td>Bailey</td></tr><tr><td>Carlson</td></tr></table></td></tr>
</table>

Rename

Purpose	Changes the name of an attribute in a relation or the name of a relation itself to something else (often used if two relations share the same attribute, by renaming one of them then we can perform operations using the two)
Syntax	$\rho_{\text{old name->new name}}(R)$
SQL Equivalent	Attributes: RENAME OBJECT [table name] COLUMN [attribute name] TO [new attribute name] Relations: RENAME OBJECT [table name] TO [new table name]
	R = <table><tr><td>sid</td><td>age</td></tr><tr><td>111</td><td>18</td></tr><tr><td>222</td><td>21</td></tr></table> $\rho_{\text{sid->sid1}}(R)$ R = <table><tr><td>sid1</td><td>age</td></tr><tr><td>111</td><td>18</td></tr><tr><td>222</td><td>21</td></tr></table>

Cartesian/Cross Product

<table>
<tr><td>Purpose</td><td>Combines all the tuples in one relation with each tuple in another relation
if one relation has r tuples and another has s tuples, the total number of tuples in their Cartesian product is r*s
If one relation has the same attribute name with the relation it is crossed it, then the attribute must be RENAMED</td></tr>
<tr><td>Syntax</td><td>R x S
where R,S are relations</td></tr>
<tr><td>SQL Equivalent</td><td>SELECT * FROM R
CROSS JOIN S;</td></tr>
<tr><td>Example</td><td>R =
<table>
<tr><td>x</td><td>y</td></tr>
<tr><td>1</td><td>3</td></tr>
<tr><td>2</td><td>4</td></tr>
</table>

S =
<table>
<tr><td>x</td><td>y</td></tr>
<tr><td>2</td><td>1</td></tr>
<tr><td>3</td><td>5</td></tr>
</table>

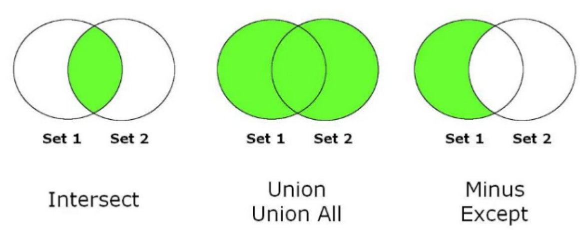
$\rho_{x\text{->}z}(S)$
$\rho_{y\text{->}w}(S)$
R x S =
<table>
<tr><td>x</td><td>y</td><td>z</td><td>w</td></tr>
<tr><td>2</td><td>3</td><td>2</td><td>1</td></tr>
<tr><td>1</td><td>3</td><td>3</td><td>5</td></tr>
<tr><td>2</td><td>4</td><td>2</td><td>1</td></tr>
<tr><td>2</td><td>4</td><td>3</td><td>5</td></tr>
</table>
</td></tr>
</table>

Set Operators

Purpose	Union: combine the data from two relations together into one; appending one table to the end of another Intersection: creating a table out of the tuples two relations have in common Set Difference: creating a table out of the tuples in the first relation that is not in the second Relations MUST have exactly the same fields (including names) in order to use set operators

Syntax	Union:
	  $R \cup S$
	Intersect:
	  $R \cap S$
	Set Difference:
	  $R - S$
	where R,S are relations

SQL Equivalent (intersect and except not valid in MySQL)	Union:
	  R UNION S
	Intersect:
	  R INTERSECT S
	Except:
	  R EXCEPT S

Example	

Natural Join

<table>
<tr><td>Purpose</td><td>Combine relations together based on related attributes between them</td></tr>
<tr><td>Syntax</td><td>R ⋈ S
Where R,S are relations</td></tr>
<tr><td>SQL Equivalent</td><td>SELECT *
FROM R
JOIN S ON [attribute name]</td></tr>
<tr><td>Example</td><td>Students =
<table>
<tr><th>name</th><th>sid</th></tr>
<tr><td>Charlie</td><td>111</td></tr>
<tr><td>Alexa</td><td>222</td></tr>
</table>
EnrolledIn =
<table>
<tr><th>sid</th><th>cid</th></tr>
<tr><td>111</td><td>CS336</td></tr>
<tr><td>111</td><td>CS214</td></tr>
<tr><td>222</td><td>CS214</td></tr>
</table>
$\pi_{name,sid,cid}$(Students ⋈ EnrolledIn) =
<table>
<tr><th>name</th><th>sid</th><th>cid</th></tr>
<tr><td>Charlie</td><td>111</td><td>CS336</td></tr>
<tr><td>Charlie</td><td>111</td><td>CS214</td></tr>
<tr><td>Alexa</td><td>222</td><td>CS214</td></tr>
</table>
</td></tr>
</table>

Functional Dependencies

Definition of Functional Dependency

Functional Dependency- X (set of attributes) determines y (set of attributes), denoyed as X→Y

e.g. Cars: (model,year) → (seats, cylinders, doors)

Definition of a Key and a Superkey

Superkey- if the closure of a set of attributes includes all attributes in the relation

Key- a superkey with a minimal number of attributes

Prime Attributes- if an attribute A belongs to at least one candidate key

To find keys given R and F: Let R = ABCD, F = {A→C, A→D, C→A}

1) Create the following table: the top row represents the location of the attribute in all functional dependencies

none/left	both	right
B	AC	D

none/left: all attributes here MUST be part of a candidate key and they are PRIME

both: attributes here could be part of a candidate key

right: no attributes here are part of a candidate key (but could be part of a superkey)

2) Compute closures of the attribute(s) in the none/left column. If these do not form a key, add the values from the both column until you get a key

Closure of B = $\frac{B+}{B}$

Closure of BA = $\frac{BA+}{BACD}$

Closure of BC = $\frac{BC+}{BCAD}$

Rules About Functional Dependencies

Armstrong Axioms

1) Reflexivity: Y ∈ X   X → Y

E.g. ABCD → A

2) Augmentation: X → Y   XZ → YZ
3) Transitivity: X → Y ∧ Y → Z   X → Z

Split-Combine Rule

X → Y ∧ Y → Z   X → YZ

Only works on the right hand side, left hand side cannot be divided

Closure of a set of attributes

Let F = {functional dependencies}

Let X ⊆ R (set of attributes)

X+ = closure of X = {A ∈ R | X → A}

= set of all attributes determined by X

Example: Let R = ABCD, F = {A → B, B → C}

A+ = {A, B, C} as A determines B which determines C, A determines itself

AD+ = {A, B, C, D}

F+ = set of all functional dependencies

Normal Forms

This link or this link can be used to find keys, a minimal cover, and normalize to 2NF, 3NF, BCNF

Lossless and Lossy Decomposition

Decomposition can be used to avoid redundancy by allowing one to calculate how to split data into two tables. It has the following goals:

- Elimination of anomalies
  - BCNF eliminates insertion, deletion, update anomalies
  - 3NF does not eliminate anomalies but reduces them
- Lossless decomposition
  - BCNF is ALWAYS lossless
- Functional Dependency Preservation
  - BCNF DOES NOT preserve functional dependencies
  - 3NF DOES preserve functional dependencies

Loseless Decomposition- you can retrieve the original table, R, from it's multiple table (decomposed) form R1, R2, etc…

The derived tables from a relation MUST share one or more attributes in common or else they cannot be combined back into the original table

Lossy Decomposition- you can retrieve part of the original table, R, from it's multiple table (decomposed) form R1, R2, etc.. (some data may be lost)

Definition of 1NF, 2NF, 3NF, and BCNF

1NF- every attribute is atomic

Violation: none (everything is in 1NF)

2NF- For all X→Y, if y is non-prime then x cannot be part of a key

Violation: [proper subset of a key] → [not prime]

3NF- For all X→Y, if y is non-prime then x must be a superkey

Violation: [not a superkey] → [not prime]

BCNF- For all X→Y, X must be a superkey

Violation: [not a superkey] → [anything]

Hierarchy of Normal Forms

BCNF ≤ 3NF ≤ 2NF ≤ 1NF

Universe of All Relations and Functional Dependencies

Cases of Normal Form

1) Every key is a singleton (has only one attribute) at least in 2NF
2) All atributes are prime at least in 3NF
3) F = { } (no functional dependencies) no violations in BCNF
4) Relations with 2 attributes in BCNF

Finding the Normal Form of a Given (R,F)

1) Find 2NF violations. If any, then strongest normal form is 1NF
2) Find 3NF violations. If any, then strongest normal form is 2NF
3) Find BCNF violations. If any, then strongest normal form is 3NF. If none, then strongest normal form is BCNF.

BCNF Decomposition

1) Check if R is in BCNF already (look for BCNF violations)
2) Use combine rules on the right hand side of functional dependencies
3) Compute the closure of A, A+, for the functional dependency A→B that violates BCNF
4) Perform, lossless decompression based on that functional dependency

    Let X = A, Y = B, and Z = all other attributes not in A or B

    Decompress into the following:

        R1 = XY

        R2 = XZ
5) Check if R1, R2 violate BCNF. If so, decompose further by repeating steps 2-4

Minimal Cover

1) Rewrite all functional dependencies so that the right hand side only has ONE attribute (singleton)

    e.g. A→BC can be written into A→C, A→B
2) Split left hand side of all functional dependencies in a way that no LHS attributes can be derived by others

    e.g. AB→C, B→A thus we can simplify this to B→C
3) Remove redundent dependencies

    Calculate the closure of all functional dependencies. If any of them have the same closure/a closure that is covered by other functional dependencies, remove them (do not use the removed functional dependency to calculate closures)
4) Use combine rules on the right hand side of functional dependencies

    e.g. A→C, A→B thus we can simplify this to A→BC

3NF Decomposition

1) Compute the minimal cover
2) Take each functional dependency and create a relation for each one (if one of the relations is covered by another relation, e.g. AC is covered by ABC, then do not include AC)

   the number of relations = size of $F+_{min}$ + 1

      e.g. $F+_{min}$ ={A→D, C→AB, E→A}

      R1 = AD

      R2 = ABC

      R3 = AE

      R4 = CE (the last relation is a candidate key which you must calculate)

Transaction Management

Definition of ACID properties

ACID is an acronym used to ensure DB reliability and deals with:

- System crashes
- Power failures
- Suspend integrity contraints while transactions are processed

Atomicity

Definition: transaction is either executed entirely or not at all

Enforced By: DBMS

Mechanism: allow transactions to be rollbacked if they cannot be entirely executed

Consistency

Definition: system/DB is in a consistent state before and after transaction (does not have to be during execution)

Enforced By: DBMS

Mechanism: detect integrity constraint violations and provide methosd to deal with them (e.g. ON DELETE CASCADE). Suspend checking of constraints in the middle of transactions

Isolation

Definition: transactions performed at the same time do not affect each other

Enforced By: Programmer

Mechanism: locks

Durability

Definition: effects of a transaction are permanent after being committed

Enforced By: DBMS

Mechanism: store to disk, make backups

Types of Schedules: Serial, Equivalent, Serializable

<u>Schedule</u>- a sequence of important steps taken by one or more transactions. These are the types of steps:

- read()

- write()
- insert()
- delete()
- commit
- abort

Serial- a schedule where one transaction is completely processed before starting the executation of another transaction

e.g. T1: R(X)   W(X)

T2:   R(Y)   W(Y)

Equivalent- when two schedules generate the same result after their executation

Serializable- a schedule that has an equivalent serial sechedule

Transactions in SQL:

```
START TRANSATION
.
.
.
COMMIT
```

Anomalies with Interleaved Execution/Concurrency Problems

Dirty Read

Definition: reading uncomitted data

Schedule Pattern:

T1: R(A)W(A)   R(B)W(B) commit

T2:   R(A)W(A) commit

Type of Problem: Consistency- T1 committed old data

Unrepeatable Read

Definition: reading different values of the same variable in different read operations even though a transaction did not update the value itself (since it should be running in isolation)

Schedule Pattern:

T1: R(A)   R(A) commit

T2:   W(A)W(A) commit

Type of Problem: Isolation- changing contents of table between two reads using write

Write Conflict/Lost Update

Definition: overwritting committed data

Schedule Pattern:

T1: W(A)   W(B) commit

T2:   W(A)W(B) commit

Type of Problem: Consistency- T1 doesn't have the most updated value of W(A)

Phantom Read

Definition: reading a variable that later gets deleted by another transaction

Schedule Pattern:

T1: R(A)   DELETE(A)
T2:   R(A)   R(A)

Type of Problem: Isolation- changing contents of table between two reads using insert/delete

Lock based concurrency control (2PL)

Two types of locks:

- S-lock- shared lock, is needed before reading data
- X-lock- exclusive lock, is needed before writing data
  - Released when the Xact has been committed

Lock Granularity: Locks can be requested/released for

- Single tuple
- Range of tuples
- Entire table

Rules: transactions must wait for a lock

1) If an Xact holds an X-lock on A, no other transaction can get an S-lock or an X-lock on that object
2) If an Xact holds an S-lock on A, another transaction can get an S-lock on A, but not an X-lock on A

deadlock- two transactions sharing the same resource are effectively preventing each other from accessing the resource. e.g.

T1: X-lock(A)   W(A)   X-lock(B)   W(B)
T2:   X-lock(B)   W(B)   X-lock(A)   W(A)

The DBMS will detect the deadlock and restart one of the Xacts using either

- Precendence graphs
- Time stamps

If an S-lock is held by an Xact then there are three choices:

1) Do not use S-locks
2) They are released immediately after reading the object
3) They are released after the Xact has committed

2PL- all locks released only after transaction has committed. Two phase locking allows only serial schedules; otherwise known as strict 2 phase locking

Isolation levels

1) Serializable

   All locks acquired are kept until the Xact has committed
   Lock indexes used by the query (holding off inserts/deletes)

2) Repeatable Read

   S-lock is needed to read and released after Xact has committed
   Prevents unrepeatable read problems

   T2 will no longer be able to write(A) b/c it will need an X-lock which it cannot get until T1 commits

Allows phantom reads
Can still perform operations like insert/delete

3) Read committed
   S-lock is needed to read and released after each read
   Prevents dirty reads
   Allows unrepeatable reads
4) Read uncomitted
   No S-locks are used
   Allows all of the anomalies (e.g. dirty read) could happen

Isolation Level	Dirty Read	Unrepeatable Read	Phantom Read
Read Uncomitted	May occur	May Occur	May Occur
Read Committed		May Occur	May Occur
Repeatable Read			May Occur
Serializable Read			

Logs (WAL)

Logs deal with aborting transactions and system crashes
Log has: OLD VALUE and NEW VALUE
Stored in disk before the actual changed data (WAL- write ahead logging)
When a transaction commits/aborts, a log record must indicate this action