Database: Data – Info – Knowledge. **Data model**: a collection of concepts for describing data. **Schema**: a description of a particular collection of data using a given a data model. **Database**: the data as stored and managed by the DBMS. **DBMS**: software package that adds functionality (MYSQL). Database System: hardware/software to support DBMS.

DB Application: **transaction**: is a sequence of database operations that are packed into a unit to which the DBMS offers certain quality guarantees (ACID). **Properties of a transaction**: ACID, atomicity, consistency, isolation, durability. A: all or nothing (unit), C: check before and after, I: concurrency control, each user does not feel the impact of others, D: log/recovery system. **Programming with transactions**: start/commit (durability) /rollback(atomicity) (all of these need to happen linearly in one function). **Integrity constrains**: constrains defined on DB schema to regulate what is consider valid instance in the DB, e.g., domain constrains.

Formal Query Languages: Relational Algebra, Relational Calculus.

RA: (1) **Relation** R (2) **Selection** σ_cond(R), **input** R, parameter cond (col op value/col op col), **output** schema same as R, instance tuple in R that satisfy condition. (3) **Projection** π_col1,col2,…,coln(R), **input** R, parameter col1,col2,…coln, **output** schema (col1,col2,…,coln), instance tuples without repetition. (4) **Set operation** R U S, R ∩ S, R-S, **input** R,S (with exactly the same schema) **output** schema same as R or S. (5) **Cross Product** RxS **input** R,S, col(R) ∩ col(S) = ∅, **output** col(R) U col(S). (6) **θ Join** R bowtie θ S (from implementation point of view it is only one operator, from a logical point of view it can be defined from other operations) **input** R,S, parameter θ -> col1 op col2, col1 in R, col2 in S, **output** schema col(R) U col(S), instance R bowtie σ S = σ_θ (RxS) (7) Natural Join R bowtie S = for each pair of attribute in R and S R.a = S.a where attributes names are the same.

RC: Declarative formal query language; basic ingredients: variable, constants, comparison operators, logical operators, quantifiers, formula. **TRC**: All variables represent tuples, constants: 1,2,"CS", Comparison <> = … logical operators AND, OR, NOT, quantifiers, forall, exists. Example {t| t ∈ Student} (get all students), {t| ∃ s ∈ Student(s.name=t.name)} (equivalent to projection in RA) (t is called de free variable not quantified)

SQL: Select (1) attr(s) (2) * (3) expression +,-,*,/,… (4) aggregate function count() max() min() avg() sum() (5) distinct(attr1,attr2,…) note: avg(age) ≠ avg(distinct(age)); **From** T1,T,2,… e,g, Enrollment as E; **WHERE** a Boolean expression; **Group By** attr(s) **Having** aggregate function (column_name) operator value. Whenever you have the group by clause, involve the attr(s) in Select.

SQL Union, Intersect, Except: (S F W) Union <All> (S F W). the default behavior in not to keep duplicates. Use keyword All to enforce keep duplicates.

SQL Nested Queries: Where Clause (1) attr IN (S F W) must return a valid set and only one column and data type comparable (2) Exists (S F W) boolean expression, return true if (S F W) is non-empty, false otherwise (3) attr = > <,..., <ALL|ANY> (S F W)

EXAMPLES:

RA: **Find the students who have never taken any course that's not required by his department**.

Students - (Students ⋈ π_{sid}(Take - $\pi_{sid,cid,term,year,grade}$ (Students ⋈ Take ⋈ RequiredCourse)))

π_{sid}(Student) – π_{sid}($\pi_{sid,cid}$(Take) – $\pi_{sid,cid}$(Students ⋈ RequiredCourses))

RC: Find the names of all students who are not in Informatics or CS.

{t| ∃ s ∈ Student(s.name=t.name ∧ ¬(s.dept='CS' ∨ s.dept='Info')}

Find the names of students who took B561 and got A

{t| ∃ s ∈ Student(s.name=t.name ∧ ∃ e ∈ Enrollment(e.sid=s.sid ∧ e.grade='A' ∧ e.cid='B561'))}

SQL: **Find the youngest students**

```
SELECT ID FROM Student WHERE age = (SELECT MIN(age) FROM Student);
```

Find the department(s) where there are more male students than female students:

```
SELECT distinct F.dept FROM (SELECT dept,count(*) as CantF From Students WHERE gender = "F" Group By dept) As F,
                            (SELECT dept,count(*) as CantM From Students WHERE gender ="M" Group By dept) As
                     WHERE M.dept = F.dept AND F.cantF<M.cantM
```

Find the students who took all courses taken by Andrew S. (Find the students who did not took any course taken by A.S.)

```
SELECT Name FROM Students As S WHERE S.name !='AS' AND NOT EXISTS
((SELECT cid FROM Students as A, Enrollment as E WHERE A.sid = E.sid and A.name = 'AS')
EXCEPT (SELECT cid FROM Enrollment as E WHERE S.sid = E.sid))
```

Find the students whose applied universities overlap with the universities applied by 0001

```
SELECT DISTINCT ID, name FROM Student, Application
WHERE ID=studentID and Univ in (
      SELECT Univ
      FROM Student, Application
      WHERE ID = 0001 and ID=studentID);
```

Find the customers who own more than three accounts (including joint accounts), compute the total amount of money in the accounts owned by each such customer

```
SELECT C.cid, SUM(A.balance)
FROM Customer C, Account A, Owner O
WHERE C.cid = O.cid AND O.aid = A.aid
GROUP BY C.cid
HAVING COUNT(A.aid) > 3
```

Find the average balance in shared accounts.

```
SELECT AVG(A.balance)
FROM Account A
WHERE A.aid IN (SELECT O.aid
                FROM Owner O GROUP BY O.aid HAVING (COUNT (DISTINCT O.cid)) > 1)
```

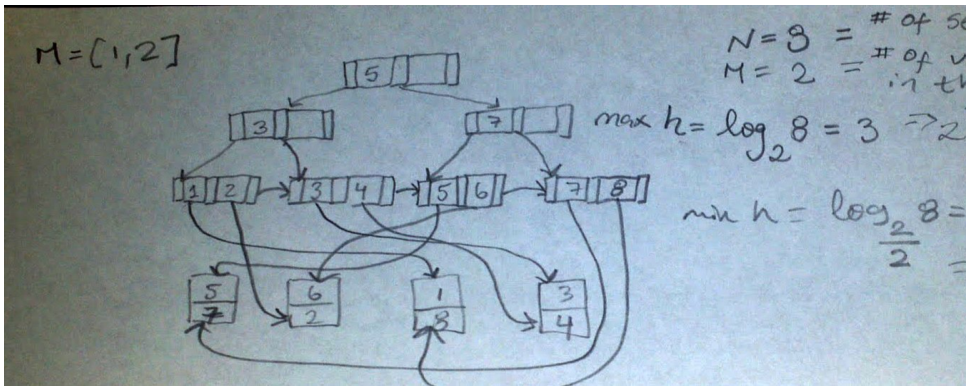
Find the accounts solely owned by a female, and list all transactions on these accounts, if any.
SELECT T.* FROM Transaction T, Owner O, Customer C
WHERE T.aid = O.aid AND O.cid = C.cid AND C.gender = “female”
AND NOT EXISTS (SELECT *
FROM Customer C1 WHERE C1.cid <> C.cid AND C1.cid = O.cid AND O.aid = T.aid)

Indexing: Index trade-off: queries fast, but updates slower. Clustering useful for range queries. Multi-attribute index: order matters. Rule of thumb: Equality selections are more selective than range selections. Clustering is useful. whenever many tuples are to be retrieved. Only one index on a given relation can be clustered**.** Many queries can be answered without retrieving any tuples from relations.

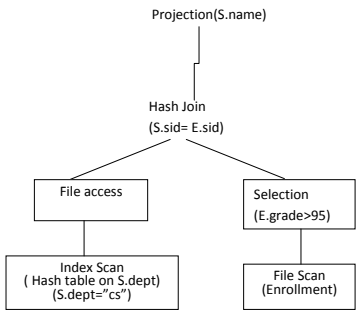
Complexity of the index operations:
Heap file: insert is O(C), just put it at the end.
B+Tree, search $h \in (log_{\frac{m}{2}}n, log_m n)$, where m is the fan-out and n number of keys. Delete will be search plus constant if we do not fall out of the fan-out, but the worst case we might need to recursively merge the nodes. Insert will be search plus constant in the case there is an empty slot, but the worst case we might need to recursively split nodes. Works for range and point queries.
Hash, search is O(C) + L, constant time (just hash the value plus linear search on the bucket). Insert: heap file O(C) + L, where H is the cost of putting the key in the hash (it might contain an extra term for the bucket overflow). Delete the same as insert. Hash is useless for range queries.

Join Algorithms
SNLJ, BNLP,PNLJ, INLJ,SMLP,HJ. Never use SNLP (cost T(R)+T(S)). If there are no indexes, use **BNLP** (cost of BNLP $P(R) + \frac{P(R)*P(S)}{B-2}$, remember to reserve two slots: one for the output and one for the inner query). **INLJ**, you make the inner table the one that has an index on the join condition (Cost P(R) * Cost of index search). **SMLP** both need to be ordered on the join condition Cost: O(|R|log|R| + |R|log|R| + |RxS|) Best Pace: P(R)+P(S). You can use it if at least the bigger relation is already sorted. **HJ**, inner table small to fit in the buffer, then construct a hash table on the join condition and then query, for each tuple in the outer relation, the inner relation using the hash. Cost P(R) * Cost of has search.

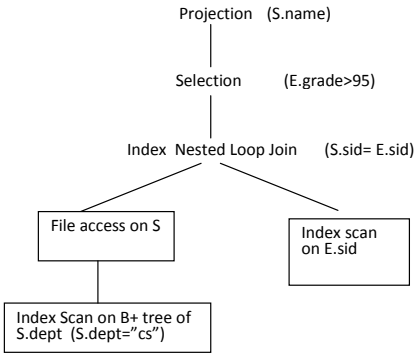
Concurrency Control: Each transaction must leave the database in a consistent state. Atomicity - commit after all, or abort after few. Anamolies: Reading uncommitted data=> WR Unrepeatable reads??=> RW
Overwriting uncommitted data - WWRemember to freeze the transaction, which are on wait list. If U cannot be upgraded, X goes to wait list, U remains granted and transaction freezes. Read, check for a write later. Rules: granted S and incoming S, then grant. Granted S and incoming U then grant. Else waiting list. Remember to specific strict 2 PL.



Case 1. Hash index on S.dept, Hash index on E.grade, no other in



Clustered B+ Tree Index on E.sid, B+Tree on S.dept.



Case 3. B+tree index on S.dept, B+tree index on E.grade, no other index available.