Relational Algebra

- Definitions
  - ↳ attribute => cols. in a relation
  - ↳ tuples => rows in a relation
  - ↳ domain => type of an attr.
  - ↳ schema => the structure of a relation
  - ↳ instance => content of a relation
  - ↳ key => attrs. that uniquely identify a tuple
- Uses SET SEMANTICS
  - ↳ No dupes + no ordering
- Operators
  - ↳ $\sigma_c R$
  - ↳ $\pi_A R$ => ~~$\pi_A(\pi_B R)$~~
  - ↳ $R \times S$
    - ↳ |output per pair of input tuples|
    - ↳ SET SEMANTICS
  - ↳ $R \bowtie S$
    - ↳ Not core, generalizes into ×
    - ↳ $R \bowtie R = R$
  - ↳ $\rho_S R$ => rename
  - ↳ $R \cup S$, $R - S$, $R \cap S$
    - ↳ Should have same schema
    - ↳ ∩ not core => $R \cap S = R - (R - S)$

Basic SQL

- Types
  - ↳ CHAR(N)
  - ↳ VARCHAR(N)
  - ↳ INT
  - ↳ DECIMAL(N,M)
    - ↳ N sigfigs, M after dec.
  - ↳ REAL
  - ↳ DOUBLE
  - ↳ TIMESTAMP
- TABLE creation
  - ↳ CREATE TABLE <name domain, ...>
    - ↳ PRIMARY KEY (attr) => can't be NULL
    - ↳ UNIQUE (attr)
    - ↳ DEFAULT <val>
  - ↳ DROP TABLE <schema>
- Data modification
  - ↳ INSERT INTO <schema> VALUES (<val> | <relation>)
  - ↳ DELETE FROM <schema> WHERE <cond>
  - ↳ UPDATE <schema> SET A=An... WHERE <cond>
- Operators
  - ↳ SELECT
    - ↳ + DISTINCT
  - ↳ FROM
  - ↳ WHERE
  - ↳ UNION, INTERSECT, EXCEPT
    - ↳ SET SEMANTICS
    - ↳ +ALL for bag semantics
  - ↳ NOT IN, IN
  - ↳ ALL, SOME
    - ↳ Ex: a > ALL R
- Subqueries
  - ↳ Nested SELECT
  - ↳ 1×1 => scalar
- Correlated subquery
  - ↳ References a table in the outer clause
  - ↳ EXISTS ( )
- Common Table Expr.
  - ↳ WITH <alias> AS <subquery>
  - ...

Advanced SQL

- Aggregate func.
  - ↳ AVG, SUM, COUNT, MIN, MAX
  - ↳ results => single tuple
  - ↳ shouldn't appear in WHERE
  - ↳ GROUP BY: forms groups where a single tuple exists for each unique attr.
    - ↳ SELECT has Agg. Funcs or unique attrs. only
  - ↳ HAVING: checks aggregate conds.
- Window Func.
  - ↳ FUNCTION (attr) OVER ( )
  - ↳ Applies agg. over all inputs
  - ↳ PARTITION BY: agg. over partition
- Case Expr.
  - ↳ CASE <attr> WHEN <cond.> THEN <expr> ELSE <expr> END
- ORDER BY / FETCH FIRST
  - ↳ ORDER BY <attr> ASC / DESC
  - ↳ <offset> FETCH FIRST <num> ROWS ONLY
    - ↳ changes bag semantics
    - ↳ LIMIT <num> OFFSET <offset>
- General SELECT
  - ↳ SELECT ...
    FROM ...
    WHERE ...
    GROUP BY ...
    HAVING ...
    ORDER BY ...
    FETCH FIRST ...
- NULLs
  - ↳ NULL + arithmetic ops = NULL
    - ↳ DIV by 0 = NULL
  - ↳ NULL = NULL
    - ↳ Logical ops w/ NULL = Unknown
  - ↳ Aggregates ignore tuples w/ NULL
    - ↳ Not COUNT(*)
    - ↳ Agg(ALL NULL) = NULL
    - ↳ COUNT(*) = 0
  - ↳ Set ops treat NULL normally
  - ↳ IS NULL or IS NOT NULL
  - ↳ COALESCE( ) => 1st non-NULL
- Outer join
  - ↳ preserves dangling tuples
  - ↳ LEFT or RIGHT
- Recursion
  - ↳ WITH RECURSIVE ancestor(child, ancestor) AS (SELECT * FROM Parent) UNION (SELECT P.child, A.ancestor FROM Parent P, Ancestor A WHERE A.parent = P.child))
    SELECT ancestor FROM Ancestor WHERE child = 'Sue',

- E/R Model
  - 2 sets: entity and relationship
    - entity: ▭
    - relationship: ◇
  - Cardinality
    - # entities participate
    - 1-to-1, 1-to-many, many-to-many
    - arrow in "1" side
    - total participation: ═
    - General notation
      - a..b, *=infinite
- Roles: designated to edges from an entity set to label the role of that entity
- Superclass and Subclass
  - Specialization and Generalization
  - Subclass inherits all attr of superclass
  - Subclass participates in the relationship of the superclass
  - Subclass may participate in its own relationship
  - Disjoint specialization vs. overlapping specialization
    - either-or vs. multiple specialization
  -  or
- Weak Entity set
  - Not enough attributes
  - ▭
  - key comes from linked entity sets
    - identifying relationship: weak entity set and owner
      - ◇

MapReduce

- Model
  - Programmer provides:
    - Map function: unit_data → (k,v), (k',v'), ...
    - Reduce function: (k,v1), (k,v2), ... → (k, aggr(v1,v2,...))
  - MapReduce handles the rest
    - Automatic data partition, distribution, and collection
    - Failure and speed-disparity handling
- Spark
  - map(): convert 1 input tuple into output tuple
  - flatMap(): convert 1 input tuple into multiple outputs
  - reduceByKey(): specify how 2 input values agg.
  - filter(): filter out tuples based on cond.

Normalization

- FD
  - U[*] => u[sid, name] => (100, Jones)
  - X → Y => no tuples can have $X_1 = X_2$, but $Y_1 \neq Y_2$
  - Trivial: (sid) => sid
- Logical implication
  - A→B, B→C => A→C
- Closure
  - F+ is the set of FDs logically implied by F
  - A^+ is set of all attributes that are functionally determined by X
    - start with $X^+ = X$ and repeat
    - X^+ is a key if X^+: {all attr}
- Lossless Decomp.: when the shared attr. are a key of one of the tables
- BCNF
  - For every non-trivial FD, $(X \to Y) \in F^+$, X contains a key
  - For any R in the schema
    - If (non-trivial X→Y holds and X does not contain a key)
      - Compute X^+
      - Decompose R into (X^+ and R − (X^+ − X))
        - X is common
        - Z is all attr. in R except X^+
    - Repeat
  - Lossless, not unique

MapReduce

- Ex)
```
lines = sc.textFile("input.txt")
words = lines.flatMap(lambda line: line.split(" "))
wordIs = words.map(lambda word: (word, 1))
wordCounts = wordIs.reduceByKey(lambda a, b: a+b)
wordCounts.saveAsTextFile("output")
```

Disk

- AccessTime = SeekTime + RotationalDelay + TransferTime
  - SeekTime: time to move disk between tracks
  - RD: depends on spin speed
    - Avg. 1 full ↻ and ↺
  - TT: time to read 1 sector
    - RPM and sectors/track
- Transfer rate: rate at which data is transferred
  - $\frac{RPM}{60} \times \frac{Sectors}{Track} \times \frac{bytes}{sector} = burst$
- Random I/O
  - More expensive for any disk
- Platter, track, cylinder, block/sector

MongoDB

- Insertion: insertX(doc(s))
  - db.books.insertOne({title:"a", attr:1})
  - db.books.insertMany({title:"c"}, {title:"b"3)
- Retrieval: findX(cond)
  - db.books.findOne({likes:100})
  - db.books.find({$and:[{likes:{$gte:10}3},{likes:{$lt:20}3}]})
- Update: updateX(cond, update)
  - update = {$set:{title:"MongoDB"}3
- Deletion: deleteX(c-c)
- Aggs.
  - aggregate([pipeline])
  - db.orders.aggregate([
    {$match:{status:"A"}3},
    {$group:{
    _id:"$cust_id",
    total:{$sum:"$amount"},
    count:{$sum:1}3
    }3
    ])
  - $sort:{$sort:{total:-1}3}
    - 1 is ASC, -1 is DESC
  - $limit:{$limit:<positive int>3}
  - $project:{$project:{_id:0,"gpa":13}
  - $unwind:{$unwind:"y"}

Files

- Spanned vs. unspanned
  - Spanned: tuples cross blocks
  - unspanned: tuples within 1 block
    - Most common
    - wastes up to ~50% of block
- Variable-len tuples
  - Reserved space - max space each tuple
    - no max waste
  - Var-len space: tight packing
    - fragmentation or insert and reshuffle
  - Slotted page: array of pointers point to tuples within blocks
- Index
  - primary/clustering: table organized by key
  - dense: 1 index entry per tuple
  - sparse: 1 index entry per block
  - secondary/nonclustering: unordered
    - dense is always index
  - overflow => more disk I/Os overtime
  - ISAM not suitable for dynamic environment

- B+tree
  - n = ptr spaces/node
  - leaf node
    - all pointers (except last) point to tuples
    - 1/2 spaces used
  - Non-leaf
    - 1/2 spaces used
  - root
    - 2 spaces used
  - Usage
    - Leaf: $\lceil \frac{(n+1)}{2} \rceil$ ptrs, $\lceil \frac{(n+1)}{2} \rceil - 1$ keys
    - non-leaf: $\lceil \frac{n}{2} \rceil$ ptrs, $\lceil \frac{n}{2} \rceil - 1$ keys
    - root: 2 ptrs, 1 key
  - Insertion
    - Leaf
      - split node in 2
      - split keys between nodes
      - 1st key of new node copied to parent + ptr to new node
    - Non-leaf
      - split in 2, move middle key to parent
    - new root
      - allocate new node
  - Deletion
    - Leaf
      - Coalesce
        - can merge w/ neighbor
        - update ptrs, delete key
      - Redistribute
        - fill node from sibling
        - update key in parent
    - Non-leaf
      - coalesce
        - merge w/ neighbor by pulling splitting key from parent
        - delete pointer to moved node
      - Redistribute
        - Overflow node by pulling splitting key down
        - Apply overflow algo
    - Root
      - delete empty node
  - n determined by node size, search key size, ptr size
- Hash index
  - Extendible Hashing: use i of $ bits from hash, dir. i as needed
    - point queries, not range
  - insertion
    - start @ i=0
    - overflow
      - i = directory i
      - double directory size by copying ptrs
      - directory i++
    - move tuples from overflowing bucket to new
    - update directory ptr
    - i++

Files

- Hash index
  - deletion
    - Merge cond.
      - Bucket
        - bucket is one stone
        - First(i-1) bits in hash key cells are same
      - Dir.
        - all bucket is < dir. i

Joins

- NLJ
  - for r ∈ R
    - for s ∈ S
      - if r.A = s.A => (r,s)
- IJ
  - create index for S.A
    - for r ∈ R
      - x = lookup index on S.A w/ r.A
      - for s ∈ x
        - (r,s)
- SMJ
  - sort R/S by A
    - i=j=1
    - while (i ≤ |R|) && (j ≤ |S|)
      - if (R[i].A = S[j].A)
        - (R[i], S[j])
        - i++, j++
      - elif (R[i].A > S[j].A) j++
      - else i++
- HJ
  - hash R tuples into G1 ... G|r| buckets
  - hash S tuples into H1 ... H|s| buckets
  - for i=1 to |r|
    - match tuples in Gi, Hi buckets

- SMJ
  - Read from R into M
  - Sort R subset
  - Generate partition of data
  - Read blocks from sorted runs into M (M-1)
  - Place smallest vals into staging block →
  - init, sort, + $\lceil \log_{M-1}(\frac{b_R}{M}) \rceil$ merges
  - Sort: $2b_R(\lceil \log_{M-1}(\frac{b_R}{M}) \rceil + 1)$
  - Join: $b_R + b_S$
  - $2b_R(\lceil \log_{M-1}(\frac{b_R}{M}) \rceil + 1) + 2b_S(\lceil \log_{M-1}(\frac{b_S}{M}) \rceil + 1) + (b_R + b_S)$
- NLJ
  - Block for stores + block for S
  - Brings as many R into M as poss
  - Scans once for each group of R
  - $b_R + \lceil b_R/(M-2) \rceil b_S$

- HJ
  - Hash into buckets
  - Hash - 2(b_R + b_S)
  - All blocks from disk into M
  - Comp with H1
  - Too big? -> hash again
    - $\lceil \log_{M-1}(\frac{b_R}{M-2}) \rceil$ times
  - $2(b_R + b_S)\lceil \log_{M-1}(\frac{b_R}{M-2}) \rceil$ blocks
- IJ
  - Pull from R, lookup, read from S
  - $b_R + |R|(c+1)$
- Summary
  - NLJ (use for small relations)
  - HJ best
    - if sorted: SMJ
    - if index: IJ
- Query opt.
  - Small on left
  - Left-deep tree
  - Deep selection

- Transactions
  - COMMIT/ROLLBACK
  - ACID
    - Atomicity: all or none
    - Consistency: consistent before, consistent after
    - Isolation: concurrent results = sequential results
    - Durability: stable after crash

- Isolation lvls
  - Dirty Read - reading val from uncommitted T
    - READ UNCOMMITTED
  - Non-Repeatable Read - Ti gets diff values on multiple reads
    - READ COMMITTED
  - Phantom - statements see parts of new transactions
    - REPEATABLE READ
  - SERIALIZABLE
  - READ ONLY = only SELECT
  - SET TRANSACTION [access_mode] ISOLATION LEVEL <level>
  - Eye of the beholding operation
- Loggings
  - <Ti, start>
    - <Ti, commit/abort>
    - <Ti, X, old-value, new-value>
  - Before commit, all records must first be disk first
  - Before tuple written back to disk, all log records flushed to disk first
    - <Ti, A, 5, 10> to disk before M:10
  - ROLLBACK => DBMS reverts to old value
  - Crash, DBMS re-executes log commands then rolls back all non-committed actions