COMP SCI 564: DBMS

Final, Fall 2022 (Lecture: AnHai Doan; Slide: AnHai Doan, Paris Koutris, R. Ramakrishnan, Evan McCarty)

Ruixuan Tu (ruixuan.tu@wisc.edu), University of Wisconsin-Madison

---

Relational algebra

- notions
  - no-bag: multiset in SQL, set (no duplicate) in relational algebra
  - schemas: $A = A, B = B, R_1(A), R_2(B)$
  - limitations: e.g., cannot compute/express transitive closure
- **5 basic operators**
  - **union** $R_1 \cup R_2$: all tuples in R_1 or R_2; $R_1, R_2, R_1 \cup R_2$ have same schema; (bag) add # occurences
  - **set difference/except** $R_1 - R_2$: all tuples in R_1 and not in R_2; $R_1, R_2, R_1 \cup R_2$ have same schema; (bag) subtract # occurences
  - **selection** $\sigma_c(R)$: returns all tuples in relation R which satisfy a condition c ($=, <, >, \textbf{and}, \textbf{or}, \textbf{not}$); output schema same as input schema; (bag) preserve # occurences
  - **projection** $\Pi_A(R)$: return certain columns, eliminates duplicate tuples; input schema $R(B)$; condition $A \subseteq B$; output schema $S(A)$; (bag) preserve # occurences, no duplicate elimination
  - **Cartesian/cross product** $R_1 \times R_2$: each tuple in R_1 with each tuple in R_2; input schemas R_1, R_2; condition $A \cap B = \emptyset$; output schema $S(A, B)$; rarely used without join; (bag) no duplicate elimination
- relations with named fields
  - **renaming** $\rho_{B_1,...,B_n}(R)$; does not change the relational instance, changes the relational schema only; input schema $R(A)$; output schema $S(B_1, ..., B_n)$
- **derived operators**
  - **intersection** $R_1 \cap R_2$: all tuples both in R_1 and in R_2; $R_1, R_2, R_1 \cap R_2$ have same schema; derivation $= R_1 - (R_1 - R_2)$
  - **join** (also, inner join and outer join)
    - **theta join** $R_1 \bowtie_\theta R_2$: a join that involves a predicate (condition θ); input schemas R_1, R_2; condition $A \cap B = \emptyset$; output schema $S(A, B)$; derivation $= \sigma_\theta(R_1 \times R_2)$
    - **natural join** $R_1 \bowtie R_2$: combine all pairs of tuples in R_1 and R_2 that agree on the join attributes $A \cap B$; input schemas R_1, R_2; output schema $S(C_1, ..., C_p)$ where $\{C_1, ..., C_p\} = A \cup B$; deviation $\sigma_{\text{agreement on join attributes}}(R_1 \times R_2)$
    - **equi-join** $R_1 \bowtie_{A=B} R_2$: natural join is a particular case of equi-join (on all the common fields); most frequently used
    - **semi-join** $R_1 \ltimes R_2$: input schemas R_1, R_2; derivation $= \Pi_A(R_1 \bowtie R_2)$
    - **division** R_1 / R_2: output contains all values a s.t. for every tuple (b) in R_2, tuple (a, b) is in R_1; input schemas $R_1(A, B), R_2(B)$; output schema $R(A)$
- **extended relational algebra**
  - **group by/aggregate** $\gamma_{X,\text{Agg}(Y)}(R)$: group by the attributes in X, aggregate the attribute in Y (SUM, COUNT, AVG, MIN, MAX); output schema: X + an extra numerical attribute

- **relational algebra experssions**, 3 notations
  - **sequences of assignment statements**: (1) create temporary relation names, (2) renaming can be implied by giving relations a list of attributes; e.g., $R_3 := R_1$ JOIN$_C$ R_2 can be written: (1) $R_4 := R_1 * R_2$, (2) $R_3 := \text{SELECT}_C(R_4)$
  - **expressions with several operators**: interpret in order, or forced order by user-inserted parentheses, from highest to lowest: (1) unary operators (select, project, rename), (2) products and joins, (3) intersection, (4) union and set difference
  - **expression trees** (usually): leaves are operands (either variables standing for relations or particular, constant relations); interior nodes are operators, applied to their child or children

Implementation of operators

- no universally best technique for most operators
- **external sorting**
  - **motivation** of sorting: data requested in sorted order; first step in bulk loading B+ tree index; eliminating duplicate copies in a collection of records, sort-merge join
  - **2-way sort with 3 buffers**: (Pass 0) read a page, sort it, write it (only 1 buffer page is used); (Pass 1, ...) three buffer page used
  - **2-way external merge sort**: each pass we r+w each page in file; N pages in file $\implies \#\ \textbf{passes} = \lceil \log_2 N \rceil + 1$; $\textbf{total cost} = 2N\left(\lceil \log_2 N \rceil + 1\right)$; idea: divide and conquer - sort subfiles, merge
  - **general external merge sort**: more than 3 buffer pages; to sort a file with N pages using B buffer pages: (Pass 0): use B buffer pages, produce $\lceil \frac{N}{B} \rceil$ sorted runs of B pages each; (Pass 1, ...) merge $B-1$ runs by sorting the first page of each sorted subset of pages; $\#\ \textbf{passes} = 1 + \lceil \log_{B-1} \lceil \frac{N}{B} \rceil \rceil$; $\textbf{total cost} = 2N * (\#\ \textbf{passes})$
  - **typical case**: if B buffer pages, a file of M pages, and $M < B * B$, then the cost of sort is $4M$. (Pass 0) create runs of B pages long, costing $2M$; (Pass 1) create runs of $B * (B-1)$ pages long: if $M < B * B$, then we are done, costing $2M$
- **joins**
  - **notion**: R is Reserves, S is Sailors; M pages for R, P_R tuples per page, N pages for S, p_S tuples per page; B buffer pages; different hash functions h_1 and h_2; cost metric: # I/Os ignoring final output costs
  - **nested loop join**
    - **tuple-based**: foreach tuple t_R in R, foreach tuple t_S in S: if t_{R_i} == t_{S_j} then join(t_R, t_S). I/O cost: $M + P_R * M * N$. $B = 2$
    - **page-based**: foreach page p_R in R, foreach page p_S in S, foreach tuple t_R in p_R, foreach tuple t_S in p_S: if t_{R_i} == t_{S_j} then join(t_R, t_S). I/O cost $M + M * N$, or if S is outer, $N + N * M$, use whichever smaller. $B = 2$
    - **block**: foreach block b_R in R, foreach page p_S in S, foreach tuple t_R in b_R, foreach tuple t_S in p_S: if t_{R_i} == t_{S_j} then join(t_R, t_S). $|b_R| = B - 2$ as 1 page as input buffer for scanning inner S, and 1 page as output buffer. R scanned once, costing M page I/Os; read S for $\lceil \frac{M}{B-2} \rceil$ times. I/O cost $M + N * \lceil \frac{M}{B-2} \rceil$. I/O cost formula: scan of outer + # outer blocks * scan of inner ($\#\ \textbf{outer blocks} = \left\lceil \frac{\#\text{ pages of outer}}{\text{blocksize}} \right\rceil$)
    - **index**: foreach tuple t_R in R, foreach tuple t_S in S where t_{R_i}==t_{S_j}: join(t_R, t_S). If there is an index on the join column of one relation (say S), can make it the inner and exploit the index. I/O cost: $M + ((M * P_R) * \textbf{cost of finding matching } S \textbf{ tuples})$. For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. $B = 2$
  - **sort-merge join** $R \bowtie_{i=j} S$
    - **procesure**: sort R and S on the join column, then scan them to do a merge, and output result tuples

    - **scan**: Advance scan of R until current R-tuple >= current S-tuple, then advance scan of S until current S-tuple >= current R-tuple; do this until current R-tuple = current S-tuple. At this point, all R-tuples with same value in R_i (current R group) and all S tuples with same value in S_j (current S group) match; output $\langle r, s\rangle$ for all pairs of such tuples. Then resume scanning R and S
    - **general cost**: R scanned once; each S group (equivalent) is scanned once per matching R tuple (with buffer hits, or nested loop, difficulty)
    - **cost if $M \leq B^2, N \leq B^2$**: sort $4M + 4N$, read in order and match $M + N$ (no duplicate/match within 1 outer page, $M * N$ as NLJ if many duplicates [$\text{output size} + M + N$ as upper bound]) by 2 buffer pages, total $5M + 5N$
    - **cost if $B = M + N$**: I/O cost B
  - **hash-join**
    - **procedure**: (1) *partition* both relations using h_1 into buckets $[1, B-1]$: R tuples could only match S tuples in same bucket; (2) *matching* tuples/h_2-partition in each partition of R and the same partition of S by hashing R by h_2 (or using block nested loop join)
    - **observation**: # partitions $k < B - 1$ (1 input buffer), $B - 2 > |\text{largest partition}|$ (1 input buffer, 1 output buffer). For uniformly sized partitions with maximal k, $k = B - 1$, $\frac{M}{B-1} < B - 2$, i.e., $B > \sqrt{M}$. Could build in-memory hashtable to speed up with more memory. If h not uniform, could apply hash-join recursively to fit some partitions which does not fit in memory
    - **I/O cost**: $3(M + N)$ (partitioning r+w both relations $2(M + N)$, matching read both relations $M + N$)
  - **general join conditions**
    - **equalities over join attributes A**: (Index NL) build index on A, or using existing indexes on a subset or an element of A. (Sort-Merge and Hash) sort/partition on combination of the columns of A
    - **inequality conditions**: (Index NL) need (clustered) B+ tree index. (Sort-Merge and Hash) not applicable. (Block NL) best method
- **other relational operations**
  - **selection** `SELECT R.C FROM Reserves R`
    - **file scan**: scan whole table, $O(M)$ I/Os
    - **index scan**: use indexes on attributes C: (hash index) $O(1)$; (B+ tree index) $\text{height} + X$ [unclustered] $X = \#\ \text{selected tuples in worst case}$, [clustered] $X = \left\lceil \frac{\#\ \text{selected tuples}}{P_R} \right\rceil$
  - **projection** `SELECT DISTINCT R.C FROM Reserves R`, $R(A)$
    - **sorting** procedure: (1) modify pass 0 of external sort to eliminate unwanted fields (M I/Os for scan, $\lceil M * \frac{C}{A} \rceil$ pages after projection and I/Os for write); (2) modify merging passes to eliminate duplicates (sorting I/Os calculated by above formula with -1 pass (pass 0 for unwanted) and pages after projection); (3) final scan (I/Os by # pages after projection)
    - **hashing** procedure: (*partitioning*) read R by 1 input buffer. for each tuple, discard unwanted fields, apply h_1 to choose a partition in $[1, B-1]$; 2 tuples from different partitions guaranteed distinct. (*duplicate elimination*) for each partition, read and build an in-memory hashtable by h_2 on all fields to remove duplicates. if partition does not fit in buffer memory, apply hash-based projection on the partition recursively
  - **set operations**
    - **intersection and Cartesian/cross product**: special cases of join
    - **union (distinct)**
      - **sorting** procedure: (1) sort both relations (on all attributes); (2) merge sorted relations eliminating duplicates

            - **hashing** procedure: (1) partition R and S by h_1; (2) build in-memory hashtable for every partition S_i (3) on that, scan corresponding partition R_i and add tuples if not duplicate
        - **set difference/except**: similar to union
    - **aggregate**
        - **without groupby**: requires scanning the relation
        - **sorting** procedure: (1) sort on group by attributes (if any); (2) scan sorted tuples, computing running aggregate; (3) when the group by attribute changes, output aggregate result; I/O cost=sorting
        - **hashing** procedure: (1) hash on group by attributes (if any) (hash entry = group attributes + running aggregate); (2) scan tuples, probe hashtable, update hash entry; (3) scan hashtable and output each hash entry; I/O cost=scan relation
        - **index** procedure
            - **without groupby**: given B+ tree on aggregate attributes in SELECT or WHERE clauses, do index-only scan
            - **with groupby**: given B+ tree on all attributes in SELECT, WHERE, and GROUPBY clauses, do index-only scan; if GROUPBY attributes form prefix of search key, tuples retrived in GROUPBY order

Query optimization

- **query plans**
    - **logical query plan**: created by the parser from the input SQL text; expressed as a relational algebra tree; each SQL query has many possible logical plans
    - **physical query plan**: goal is to choose an efficient implementation for each operator in the RA tree; each logical plan has many possible physical plans
        - **transformed**: access path selection for each relation (scan or index); implementation choice for each operator (e.g., nested loop join, hash join); scheduling decisions for operators (pipelined or batch)
- **execution**
    - **pipeline**: tuples generated by an operator are immediately sent to the parent (used whenever possible)
        - **benefits**: no operator synchronization issues; no need to buffer tuples between operators; **no r+w intermediate data from disk**
    - **batch/materialize**: write the intermediate result before we start the next operator (which read the result)
- **query optimization process**: (1) identifies candidate equivalent relational algebra trees (i.e., logical query plan); (2) for each relational algebra tree, it finds the best annotated version (using any available indexes) (i.e., physical query plan); (3) chooses the best/cheapest overall plan by estimating the I/O cost of each plan
    - **System R optimizer**: *cost estimation* for cost of operations and result sizes, by approximate with statistics, considering CPU + I/O costs; to prune large *plan space*, only consider the space of left-deep plans and avoid cartesian products
- **relational algebra tree transformation** on physical plan enumeration
    - **pushing down** (execute as early as possible in query plan)
        - **selections**: always possible to change the order through projections, joins, other selections
        - **projections**: through selections, joins
        - **reason**: fewer tuples in intermediate steps of plan
        - **note**: unable to use the index of a column after pushing a selection down
    - **join reordering** by $R \bowtie S \bowtie T \bowtie U$
        - **properties**: (*communitativity*) $R \bowtie S \equiv S \bowtie R$; (*associativity*) $(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)$; can reorder in any way (exponentially many)
        - **left-deep join**: $((R \bowtie S) \bowtie T) \bowtie U$; benefit to focus: allow pipeline; $n!$ possible trees
        - **right-deep join**: $R \bowtie (S \bowtie (T \bowtie U))$; $n!$ possible trees

        - **bushy join**: $(R \bowtie S) \bowtie (T \bowtie U)$; $\frac{(2n-2)!}{(n-1)!}$ possible trees
- **cost estimation** of query plan
    - must estimate **cost** of each operation in plan tree; depends on input cardinalities; algorithm cost (previously)
    - must also estimate **size** of result for each operation in tree; use information about the input relations; for selections and joins, assume independence of predicates
- **system catalog** updated periodically (everytime is expensive)
    - **statistics**: # tuples and # pages for each relation; # distinct key values and # pages for each index; index height, low/high key values for each tree index
    - histograms for some values are sometimes stored

Transaction management

- **motivation**: recovery, durability, concurrency, or in all to avoid inconsistency
- **transaction**: a sequence of SQL statements that you want to execute as a single atomic unit; `BEGIN TRANSACTION; {SQL} COMMIT;` or `START TRANSACTION {SQL} END TRANSACTION`, use `ROLLBACK` for `COMMIT` to abort
    - without: execute a transaction half way (e.g., app crash); that can leave app in an inconsistent state
- **ACID properties**: **a**tomic, **c**onsistent, **i**solation, **d**urable
    - **atomic**: all actions in the transaction happen, or none happen. if a transaction crashes half way, then remove its effect
    - **consistent**: a database in a consistent state will remain in a consistent state after the transaction
    - **isolation**: the execution of a transaction is isolated from other (possibly interleaved) transaction. if two users run transactions concurrently, they should not interfere with each other
    - **durable**: once a transaction commits, its effects must persist
- **implementation**: DB ensures ACID by using locks and crash recovery. User App must be structured as executing transactions on a database

Recovery

- **types of failures**
    - **wrong data entry**: prevent by having constraints in the database; fix by data cleaning
    - **disk crashes**: prevent by using redundancy (RAID, archive); fix by using archives
    - **system failures**: most frequent (e.g., power); use recovery by log (as internal state is lost)
- **log**: a file that records every single action of the transaction
    - an append-only file containing log records
    - multiple transactions run concurrently, log records are interleaved
    - after a system crash, use log to: redo/undo some transaction that did not commit
- **elements**: assumes that the database is composed of elements (usually 1 element = 1 block, can be = 1 record or = 1 relation); assumes each transaction r/w some elements
- **primitive operations of transactions**
    - `INPUT(X)`: read element `X` to memory buffer
    - `READ(X, t)`: copy element `X` to transaction local variable `t`
    - `WRITE(X, t)`: copy transaction local variable `t` to element `X`
    - `OUTPUT(X)`: write element `X` to disk
- **undo logging**
    - **log records**

        - `<START T>`: transaction `T` has begun
        - `<COMMIT T>`: `T` has committed
        - `<ABORT T>`: `T` has aborted
        - `<T,X,v>`: `T` has updated element `X`, and its *old* value was `v`
    - **rules**
        - If `T` modifies `X`, then `<T,X,v>` must be written to disk before `X` is written to disk
        - If `T` commits, then `<COMMIT T>` must be written to disk only after all changes by T are written to disk (no need to undo)
        - `OUTPUT`s are done *early* (before `COMMIT`)
    - **recovery** after system crash
        - **procedure**: (1) decide each transaction `T` whether completed: (complete) `<START T> ... <COMMIT T>`, `<START T> ... <ABORT T>`; (incomplete) `<START T> ......`. (2) undo all modifications by *incompleted* transactions
        - **read log from end; cases**: (`<COMMIT T>`/`<ABORT T>`) mark `T` as completed; (`<T,X,v>`) if `T` not completed then write `X=v` to disk, else ignore; (`<START T>`) ignore
        - all undo commands are **idempotent**: if we perform them a second time, no harm is done (e.g., crash during recovery)
        - **stop reading**: until beginning of log file, or (better) use **checkpointing**
        - **recovery with nonquiescent checkpointing** procedure: (1) look for the last `<END CKPT>`, undo all uncommitted transactions along the way; (2) stop until the corresponding `<START CKPT>`
    - **checkpointing**
        - **checkpoint** the database periodically: (1) stop accepting new transactions; (2) wait until all curent transactions complete; (3) flush log to disk; (4) write a log record, flush; (5) resume transactions
        - **nonquiescent checkpointing**: checkpoint while database is operational (not freezing DB)
            - **procedure**: (1) write a `<START CKPT(T1, ..., Tk)>` where `T1, ..., Tk` are all active transactions; (2) continue normal operation; (3) when all of `T1, ..., Tk` have completed, write `<END CKPT>` (ensures the system did not crash and the checkpoint terminated)
- **redo logging**
    - **log records** 1 change: `<T,X,v>`: `T` has updated element `X`, and its *new* value is `v`
    - **rules**
        - If `T` modifies `X`, then both `<T,X,v>` and `<COMMIT T>` must be written to disk before `X` is written to disk
            - If `<COMMIT T>` is not seen, `T` definitely has not written any of its data to disk (no dirty data)
        - `OUTPUT`s are done *late* (after `COMMIT`)
    - **recovery** after system crash
        - **procedure**: (1) decide each transaction `T` whether completed (same as undo logging); (2) read log from the beginning, redo all updates of *committed* transactions
        - **nonquiescent checkpointing** procedure: (1) write a `<START CKPT(T1, ..., Tk)>` where `T1, ..., Tk` are all active transactions; (2) flush to disk all blocks of committed transactions (dirty blocks), while continuing normal operation; (3) when all blocks have been written, write `<END CKPT>`
        - **recovery with nonquiescent checkpointing** procedure: (1) look for the last `<END CKPT>`; (2) redo all committed transactions that are listed in and starting after this `<START CKPT ...>`
- **undo/redo logging**
    - **log records** 1 change: `<T,X,u,v>`: `T` has updated element `X`, its old value was `u`, and its new value is `v`
    - **rule**
        - If `T` modifies `X`, then `<T,X,u,v>` must be written to disk before `X` is written to disk
        - Free to `OUTPUT` *early* or *late*

- **recovery** procedure: (1) redo all committed transaction, top-down; (2) undo all uncommitted transactions, bottom-up

Normalization

- **types of anomalies**
  - **redundancy**: repetition of data
  - **update anomalies**: update one item and forget others = inconsistencies
  - **deletion anomalies**: delete many items, delete one item, loose other information
  - **insertion anomalies**: cannot insert one item without inserting others
- **good design**: (1) start with original db schema R; (2) transform it until we get a good design R^*
  - **desirable properties of R^*/schema refinement**: minimize redundancy; avoid info loss; preserve dependencies/constraints; ensure good query performance (can be conflicting)
- **normal forms**: transform R to R^* in some of normal forms
  - **motivation**: recognize a good design R^*; transform R into R^*; using R directly causes anomalies
  - **examples**: Boyce-Codd or 3.5NF (focus), 3NF (FD preserving), 1NF (all attributes are atomic) normal forms
  - If R^* is in a normal form, then R^* is guaranteed to achieve certain good properties
  - **procedure**: (1) take a relation schema; (2) test it against a normalization criterion; (3) if it passes, fine! maybe test again with a higher criterion; (4) if it fails, decompose into smaller relations; each of them will pass the test; each can then be tested with a higher criterion
- **functional dependencies**
  - **definition** $A \to B$ (A functionally determines B): if two tuples agree on attributes $A_1, \ldots, A_n$ as A, then they must also agree on attributes $B_1, \ldots, B_m$ as B
  - **properties**: a form of constraint (in schema); finding them is part of DB design; used heavily in schema refinement
  - **checking** $A \to B$: (1) erase all other columns; (2) check if the remaining relation is many-one (*functional* in math)
  - **creating schema**: list all FDs we believe valid; FDs should be valid on *all* DB instances conforming the schema
- **relation keys**
  - **key** of relation R: a set of attributes that functionally determines all attributes of R (certain FDs are true); none of its subsets determines all attributes of R
  - **superkey**: a set of attributes that contains a key; including a key itself
  - **rules for finding key of relation** from: (entity set) the set of attributes which is the key of the entity set; (many-many) the set of all attribute keys in the relations corresponding to the entity sets
  - **trivial**: An FD $X \to A$ is called *trivial* if the attribute A belongs in the attribute set X
- **Armstrong's Axioms** on sets of attributes like $A = \{A_i\}_{i=1}^{i=n}$ (other sets could of different sizes)
  - **basic rules**: (*reflexivity*) $A \to$ **a subset of** A; (*augmentation*) if $A \to B$ then $AC \to BC$; (*transitivity*) if $A \to B$ and $B \to C$ then $A \to C$
  - **additional rules**: (*union*) if $X \to Y$ and $X \to Z$ then $X \to YZ$; (*decomposition*) if $X \to YZ$ then $X \to Y$ and $X \to Z$; (*pseudo-transitivity*) if $X \to Y$ and $YZ \to U$ then $XZ \to U$
  - **closure of FD set** S as S^+: all FDs logically implied by S
    - **procedure** of inference: (1) $S^+ \leftarrow S$; (2) loop: (2-1) foreach f in S apply reflexivity and augmentation rules, (2-2) add new FDs to S^+, (2-3) foreach pair of FDs in S apply the transitivity rule, (2-4) add newe FDs to S^+; (3) finish when S^+ does not change any further
  - **closure of attribute set** A as A^+: (1) $A^+ \leftarrow A$; (2) loop: if $B \to C$ is in S and B are all in X and C is not in X then add C to A^+; (3) finish when A^+ does not change any further

    - **usage**: (test if X a superkey) check if X^+ contains all attributes of R; (check if $X \to Y$ holds) check if Y is contained in X^+
    - another way to **compute FD closure S^+**: (1) foreach subset of attributes X in relation R: compute X^+; (2) foreach subset of attributes Y in X^+: output FD $X \to Y$
- **relational schema/logical design**: (*conceptual model*) ER diagram; (*relational model*) create tables, specify FDs, find keys; (*normalization*) use FDs to *decompose* tables for better design
- **relation decomposition**
  - **in general**: decompose $R(A)$ into $R_1(B)$ and $R_2(C)$ s.t. $B \cup C = A$ and R_1 is projection of R on B and R_2 is projection of R on C
  - **lossless** (desirable property #2): a decomposition is lossless if we can recover ($R(A, B, C) \to R_1(A, B), R_2(A, C) \to R'(A, B, C)$, $R' = R$ not larger)
  - another definition of **lossless decomposition**: decompositions which produce only lossless joins
  - **lossy join**: if you decompose a relation schema, then join the parts of an instance via a natural join, you might get more rows than you started with
  - **FD preserving** (desirable property #3): given a relation R and a set of FDs S and decomposition $R \to R_1, R_2$, suppose R_1 has a set of FDs S_1, R_2 has a set of FDs S_2, we say the decomposition is *FD preserving* if by enforcing S_1 over R_1 and S_2 over R_2 we can enforce S over R
    - **not FD preserving for $X \to Y$**: when a relation is decomposed, the X of ends up only in one of the new relations and the Y ends up only in another
- **BCNF**
  - **definition**: a relation R is in BCNF iff: whenever there is a nontrivial FD $A \to B$ for R then A is a superkey for R
  - equivalent **definition**: for every attribute set X in R, either $X^+ = X$ or $X^+ = \text{all attributes}$
  - **decomposition** procedure: (1) find a FD that violates the BCNF condition $A \to B$ (heuristics: choose largest B); (2) decompose A and B to R_1, A and remaining attributes to R_2 (any 2-attribute relation is in BCNF); (3) continue until no BCNF violations left
  - **properties** of BCNF decomposition: removes all redundancy based on FD; is lossless-join; is not always FD preserving