Final Review

courtesy of Joe Hellerstein for some slides

Jianlin Feng
School of Software
SUN YAT-SEN UNIVERSITY

Topics Covered

- Relational Model
- Disks, Files, Buffers, Indexes
- Languages: Algebra, SQL
- Query Processing: Sorting, Hashing, Join Algorithms
- Query Optimization
- Schema Refinement and Normalization
- Concurrency Control
- Crash Recovery

Overview

- Purpose of this course: give students both
  - An understanding of what systems do, why we use them, and how to use common databases efficiently,
  - and an understanding of how databases work internally.

Introduction

- What are databases?
- Data models
- What does a DBMS provide that the OS does not?
  - Levels of Abstraction, Data Independence, Concurrency Control, Crash Recovery, etc.

Introduction (Contd.)

- schemas & data independence
  - conceptual schema
  - physical schema
  - external schema (view)
  - logical & physical data independence

The Relational Data Model Basics

- Components of the model:
  - Relations, Attributes, Tuples
- SQL Data Definition Language
- Integrity Constraints
  - how do they come into being?
  - understand what you can learn from schema vs. instance!
- *Referential Integrity*
  - a state which holds when all foreign key constraints are enforced.

Relational Model (Contd.)

- Keys, Primary Keys, Foreign Keys, Candidate Keys
- *Foreign key* : Set of fields in one relation that is used to `refer' to a tuple in another relation.
  - Must correspond to primary key of the second relation.
  - Like a `logical pointer'.
- A set of fields is a *key* for a relation if :

1. No two tuples can have same values in all key fields, and
2. This is not true for any subset of the key.

  - Part 2 false? A *superkey*.
  - If there's >1 key for a relation, all are *candidate keys.* One of the candidate keys is chosen to be *primary key*.
- E.g., *sid* is a key for Students. (What about *name*?) The set {*sid, gpa*} is a superkey.

Memory Management

- Hierarchy of storage: RAM, Disk, Tape
- Advantages/disadvantages of different types of storage .
- Buffer management
  - You should know basics such as
    - Understand data structures required
    - notions of *replacement*, *dirty* pages, *pinning* pages
  - Understand different replacement policies
    - LRU, MRU, CLOCK
    - be able to simulate each, understand pros, cons!

Memory Management (Contd.)

- Organizing records in pages
  - Fixed & variable-length fields in tuples
    - 2 alternatives for variable-length fields
  - Fixed & variable-length tuples on pages
    - know what a RID is, how it interacts with page layout
    - know details of “slotted page” with slot directory
- Organizing pages in files

File Organization

- Different file organizations: heap files, sorted files, hashed files
  - be able to compute costs of operations over each!
- Access costs for different organizations

File Organization: Indexes

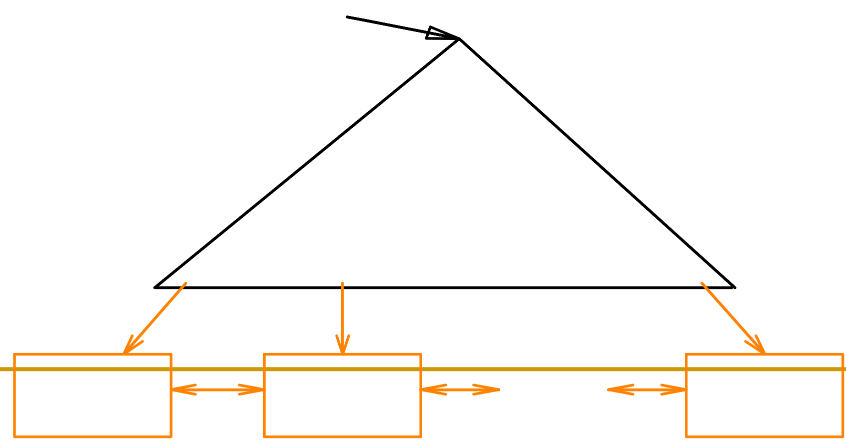
- Understand search keys (vs. key constraints!)
- 3 alternatives for data entries
  - Data record with key value **k**
  - <**k**, rid of data record with search key value **k**>
  - <**k**, list of rids of data records with search key **k**>
  - Choice of alternative orthogonal to indexing technique!

Tree Structured Indexes

- Trees do *range* and *equality* search
- ISAM, rules for adding and removing entries
- B-Trees, rules for adding entries
  - You do not have to do “coalescing” on B+-tree deletion

B+ Tree Structure

- Each node contains **d** <= $\underline{m}$ <= 2**d** entries (index or data)
  - The parameter **d** is called the *order* of the tree.
  - Each internal node contains *m* index entries: <key, page id>.
  - Each leaf node contains *m* data entries: <key, record or record id>
- The ROOT node contains between 1 and 2**d** index entries.
  - It is a leaf or has at least two children.
- Each path from the ROOT to any leaf has the same length.
- Supports equality and range-searches efficiently.



External Sorting, Hashing

- How to sort any file using 3 memory Pages
- How to sort/hash in as few passes given some amount of memory
- Relationship between buffers of memory, size of file, and number of passes to merge
- Duality of sort and hash
- Application to duplicate elimination, group by

Relational Algebra

- Query language operating on relations
- Know the operators!
  - σ, π, $\times$, $\cup$, $\cap$, $-$, ρ,
  - know schemas of output relations
  - know varieties of joins
    - (conditional vs. equi vs. natural), division
  - be able to express complex operations in terms of simple ones
- Use relational algebra to express queries written in English, and vice versa.

SQL

- DDL: “Create Table”
- DML: Delete From, Insert Into, Update
- Basic Query:

  select <targets>

  from <relations>

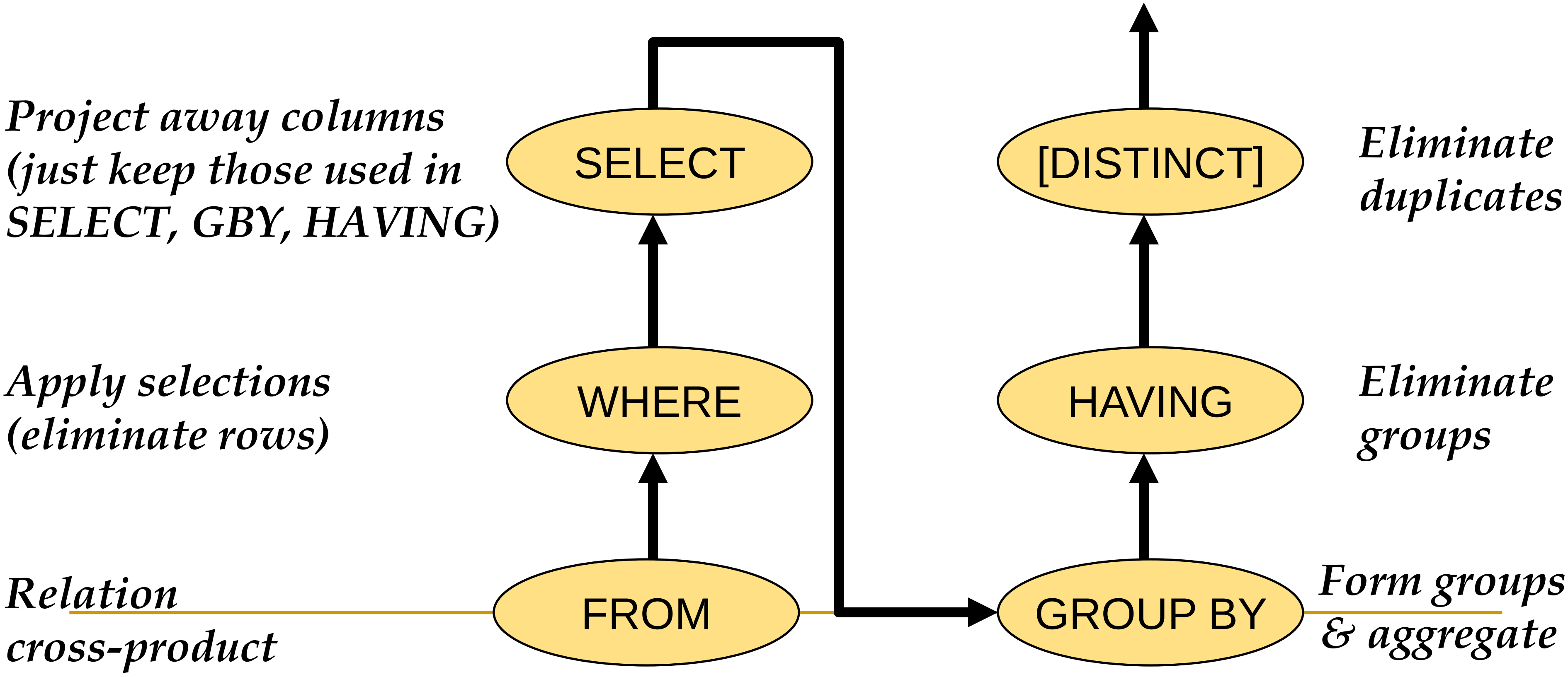
  where <qualification>
- Use of Distinct clause
- Set operations: Union, Except, Intersect

SQL Query Language

- basic queries
- conceptual evaluation
- range variables
- expressions
- strings
- Union, Intersect, In, Except, Exists
- nested queries, correlated and not
- set comparison
- Aggregation
  - operators: Count, Avg, Any
  - Group By, Having

Conceptual SQL Evaluation

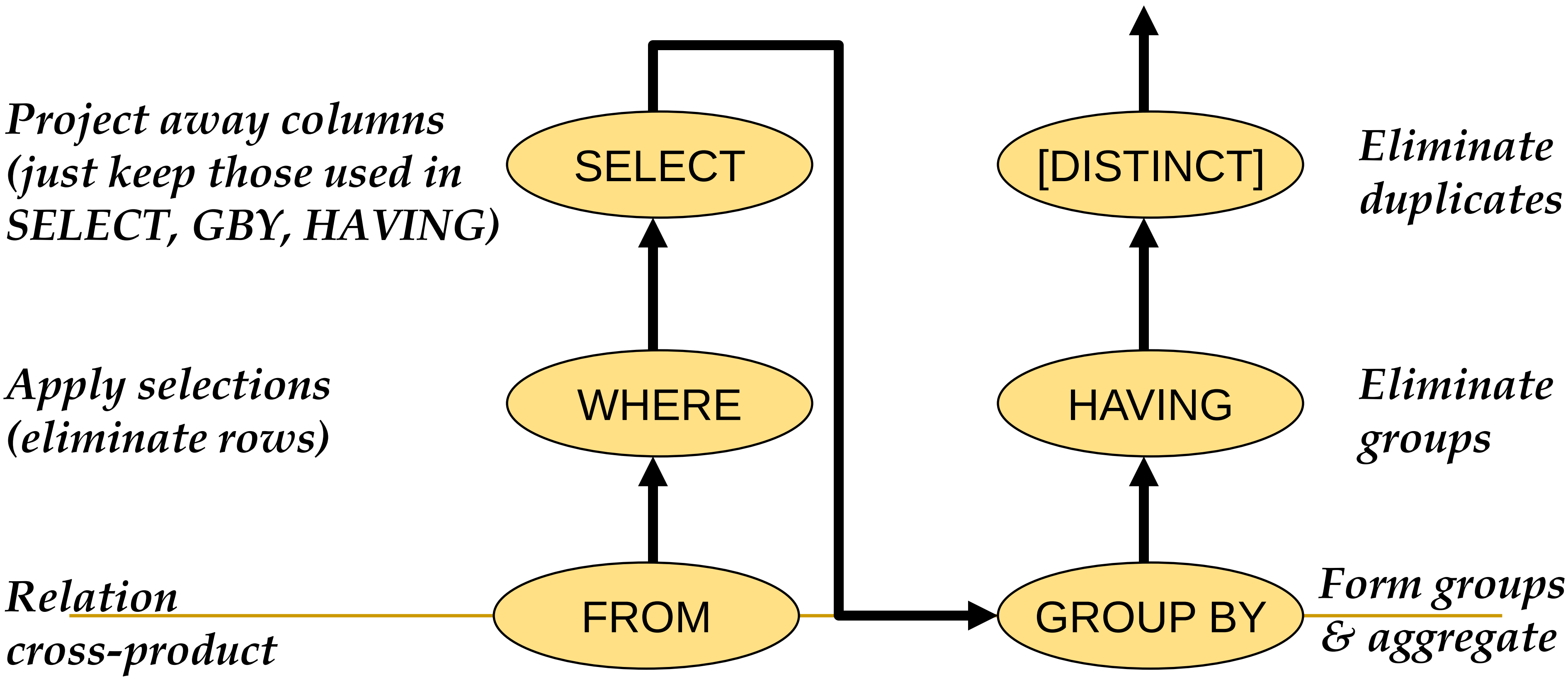
SELECT	[DISTINCT] *target-list*
FROM	*relation-list*
WHERE	*qualification*
GROUP BY	*grouping-list*
HAVING	*group-qualification*



Conceptual SQL Evaluation

SELECT [DISTINCT] *target-list*
FROM *relation-list*
WHERE *qualification*
GROUP BY *gr*
HAVING *gr...ion*

ORDER BY



SQL Continued

- Aggregation
  - Count, Sum, Avg, Max, Min
  - Group By, Having clauses
- Nested Queries
  - in Where or From Clause
  - set comparison:
    - In, Exists, Unique
    - *op* Any, *op* All
  - correlated vs. uncorrelated

Implementation of Relational Operators

- Important Operators:
  - Selection
  - Projection
  - Set operations
    - Set-Difference, Union
  - Aggregation
  - Join
- Understand cost estimation, selectivity

Selection

- with no index, scan entire relation
- with clustered index, use index
- with unclustered index, sort rids
- hash index only good for equality selection
- with multiple selection conditions, either
  - scan entire table testing all conditions
  - use index on most restrictive condition first, scan result for other conditions
  - use index for each condition, do set intersection on RIDs

Projection

- Hard part: removing duplicates (if necessary)
- Can remove duplicates by sorting or hashing
- If index contains projected attr, can do index-only scan

Set Operations & Aggregation

- Set operations:
  - Intersection, cross product treated like joins
  - Union (Distinct) and Set Difference both involve finding duplicates between two sets, treat similar to Project
- Aggregation
  - without Group By, must scan entire relation
  - with Group By, could sort, then scan
    - Or…?

Joins

- Two relations: inner N and outer M
- Simple Nested Loops:
  - for each outer tuple, scan inner for matches
  - cost:M + #tuples in M * N
- Paged Nested Loops
  - for each page in M, scan inner for matches to any tuple in that M page
  - cost: M + M*N
- Blocked Nested Loops
  - like paged, except put as much of M in memory as possible, leaving 1 page for N and 1 page for output
  - cost: M + (M/(block size)) * N

Joins (Contd.)

- Indexed Nested Loops
  - for each tuple in M, use index to find matches in N
  - cost: M + #tuples in M * cost to use index to get tuples
- Sort-Merge Join
  - Sort each table, merge finding like values,
  - can be bad if many duplicates
  - cost: M log M + N log N + M + N
- Hash Join
  - partition both relations into buckets, read in one bucket from M at a time, match with elements from same bucket in N
  - cost: partioning 2*(M + N) plus matching (M + N)

Query Optimization

- Some operations are commutative, associative
- May change the order of many operations in query
- Dramatic changes in cost depending on operation order
- “Query Plan” - a tree of operations indicating operation implementations and order
- Ideally find optimal plan
- In reality avoid terrible plans

Optimizer Implementation

- need iterator interface so each operation passes tuples on to the next
- need cost estimator to determine costs of different plans
  - Reduction Factor of operations
- need statistics and catalogs to estimate costs

System R Optimizer in Action

- convert SQL to relational algebra
- find alternate plans
  - for each relation, consider all access paths
  - for multiple relations, consider different join algorithms, access paths
  - for join orders, only consider left-deep trees

Functional Dependencies (FDs)

- An FD $X \rightarrow Y$ holds over relation schema R if, for *every allowable instance r* of *R*:

  $$t1 \in r,\ \ t2 \in r,\ \ \pi_X(t1) = \pi_X(t2)$$
  $$\textit{implies}\quad \pi_Y(t1) = \pi_Y(t2)$$

  (*t1*, *t2* are tuples; *X*, *Y* are *sets* of attributes)
- In other words: $X \rightarrow Y$ means
  - Given any two tuples in r,
    if the X values are the same,
    then the Y values must be the same.
    (but not vice versa)
- Read "$\rightarrow$" as "*determines*"

Problems Due to R $\rightarrow$ W

S	N	L	R	W	H
123-22-3666	Attishoo	48	8	10	40
231-31-5368	Smiley	22	8	10	30
131-24-3650	Smethurst	35	5	7	30
434-26-3751	Guldu	35	5	7	32
612-67-4134	Madayan	35	8	10	40

Hourly_Emps

- *Update anomaly*: Can we modify W in only the 1st tuple of SNLRWH?
- *Insertion anomaly*: What if we want to insert an employee and don't know the hourly wage for his or her rating? (or we get it wrong?)
- *Deletion anomaly*: If we delete all employees with rating 5, we lose the information about the wage for rating 5!

Rules of Inference

- Armstrong's Axioms (X, Y, Z are <u>sets</u> of attributes):
  - *<u>Reflexivity</u>*: If $X \supseteq Y$, then $X \rightarrow Y$
  - *<u>Augmentation</u>*: If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z
  - *<u>Transitivity</u>*: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

- S*ound* and *complete* inference rules for FDs!
  - using AA you get *only* the FDs in F+ and *all* these FDs.

- Some additional rules (that follow from AA):
  - *Union*: If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
  - *Decomposition*: If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$

Attribute Closure

- Computing closure F^+ of a set of FDs F is hard:
  - exponential in # attrs!
- Typically, just check if $X \rightarrow Y$ is in F^+. Efficient!
  - Compute *attribute closure* of X (denoted X^+) wrt F.
    $X^+ =$ Set of all attributes A such that $X \rightarrow A$ is in F^+
    - $X^+ := X$
    - Repeat until no change:
      if $U \rightarrow V \subseteq F$, $U \subseteq X^+$, then add V to X^+
  - Check if Y is in X^+
  - Approach can also be used to find the keys of a relation.
    - If $X^+ = R$, then X is a superkey for R.
    - Q: How to check if X is a "candidate key"?

Boyce-Codd Normal Form (BCNF)

- Reln R with FDs F is in BCNF if, for all $X \rightarrow A$ in F+
  - $A \subseteq X$ (called a trivial FD), or
  - X is a superkey for R.
- In other words: "R is in BCNF if the only non-trivial FDs over R are key constraints."

Decomposition of a Relation Scheme

- How to normalize a relation?
  - *decompose* into multiple normalized relations
- Suppose *R* contains attributes *A1 … An.*
  A *decomposition* of R consists of replacing R by two or more relations such that:
  - Each new relation scheme contains a subset of the attributes of R, and
  - Every attribute of R appears as an attribute of at least one of the new relations.

Lossless Join Decompositions

- Decomposition of R into X and Y is *lossless-join* w.r.t. a set of FDs F if, for every instance r that satisfies F:

$$\pi_X(r) \bowtie \pi_Y(r) = r$$

- It is always true that $r \subseteq \pi_X(r) \bowtie \pi_X(r)$
  - In general, the other direction does not hold!
  - If it does, the decomposition is lossless-join.
- Definition extended to decomposition into 3 or more relations in a straightforward way.
- *It is essential that all decompositions used to deal with redundancy be lossless! (Avoids Problem #1)*

Lossy Decomposition (example)

A	B	C
1	2	3
4	5	6
7	2	8

$A \rightarrow B; C \rightarrow B$

⇨

A	B
1	2
4	5
7	2

B	C
2	3
5	6
2	8

A	B
1	2
4	5
7	2

$\bowtie$

B	C
2	3
5	6
2	8

=

A	B	C
1	2	3
4	5	6
7	2	8
1	2	8
7	2	3

Lossless Decomposition (example)

A	B	C
1	2	3
4	5	6
7	2	8

A	C
1	3
4	6
7	8

B	C
2	3
5	6
2	8

$A \rightarrow B$; $C \rightarrow B$

A	C
1	3
4	6
7	8

$\bowtie$

B	C
2	3
5	6
2	8

$=$

A	B	C
1	2	3
4	5	6
7	2	8

But, now we can't check $A \rightarrow B$ without doing a join!

Dependency Preserving Decompositions (Contd.)

- Decomposition of R into X and Y is ***dependency preserving*** if $(F_X \cup F_Y)^+ = F^+$
  - i.e., if we consider only dependencies in the closure F^+ that can be checked in X without considering Y, and in Y without considering X, these imply all dependencies in F^+.
  - (just the formalism of our intuition above)
- Important to consider F^+ in this definition:
  - ABC, $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow A$, decomposed into AB and BC.
  - Is this dependency preserving? Is $C \rightarrow A$ preserved?????
- Note: F^+ contains $F \cup \{A \rightarrow C, B \rightarrow A, C \rightarrow B\}$, so…
  - $F_{AB} \supseteq \{A \rightarrow B, B \rightarrow A\}$; $F_{BC} \supseteq \{B \rightarrow C, C \rightarrow B\}$
  - So, $(F_{AB} \cup F_{BC})^+ \supseteq \{C \rightarrow A\}$

Third Normal Form (3NF)

- Reln R with FDs *F* is in 3NF if, for all $X \rightarrow A$ in F^+
  - $A \in X$ (called a *trivial* FD), or
  - X is a superkey of R, or
  - A is part of some candidate key (not superkey!) for R.
    (sometimes stated as "A is *prime*")
- *Minimality* of a candidate key is crucial in third condition above!
- If R is in BCNF, obviously in 3NF.
- If R is in 3NF, some redundancy is possible. It is a compromise, used when BCNF not achievable (e.g., no ``good'' decomp, or performance considerations).
  - *Lossless-join, dependency-preserving decomposition of R into a collection of 3NF relations always possible.*

Minimal Cover for a Set of FDs

- *Minimal cover* G for a set of FDs F:
  - Closure of F = closure of G.
  - Right hand side of each FD in G is a single attribute.
  - If we modify G by deleting an FD or by deleting attributes from an FD in G, the closure changes.
- Intuitively, every FD in G is needed, and ``*as small as possible*'' in order to get the same closure as F.
- e.g., $A \rightarrow B$, $ABCD \rightarrow E$, $EF \rightarrow GH$, $ACDF \rightarrow EG$ has the following minimal cover:
  - $A \rightarrow B$, $ACD \rightarrow E$, $EF \rightarrow G$ and $EF \rightarrow H$
- M.C. implies Lossless-Join, Dep. Pres. Decomp!!!
  - (in book)

Concurrency Control

- Transaction: basic unit of operation
  - made up of reads and writes
- Goal: ACID Transactions
- A & D are provided by Crash Recovery
- C & I are provided by Concurrency Control
- Bottom line: reads and writes for various transactions MUST be ordered such that the final state of the database is the same as *some serial ordering of the transactions*

Approaches to Concurrency Control

- 2PL - all objects have Shared and eXclusive locks
  - once one lock is released, no more locks may be acquired
  - Strict 2PL: don't release locks until commit time
- Locking issues
  - must either prevent or detect deadlock
  - may want multiple granularity locks (table, page, record) using IS, IX, SIX, S, X locks *(check compatibility matrix!)*
  - locking in B-trees usually not 2PL
  - phantom problem: locking all records of a given criteria (e.g., age > 20)

Crash Recovery

- ACID - need way to ensure A & D
- We studied approach of Aries system
- Buffer management Steal, no Force
- Every Write to a page is first logged in WAL
  - log record is in stable storage before data page on disk
  - log record has Xact#, before value, after value
- Checkpoints record which pages dirty, which XActs running

Transaction Commit

- write Commit record to log
- flush log tail to stable storage
- remove Xact from Xact table
- write End record to log

Transaction Abort

- write Abort record to log
- go back through log, undoing each write (and add CLR to log)
- when done, write End record to log

Crash Recovery - 3 phases

- Analysis: starting from checkpoint, go forward in the log to see:
  - what pages were dirty
  - what transactions were active at time of crash
- Redo: start from oldest log record (oldest recLSN) that wrote to a dirty page, and redo all writes to dirty pages.
- Undo: start at the end of the log (time of crash), work backward undoing all writes made by transactions that were active at time of crash