Revision

Overview: Database Design

- Data models: ER, Relational Data Model and their mapping
- Relational Algebra: be able to use relational algebra to answer question.
- Database Languages: SQL, PLpgSQL (final exam: need be able to determine yes or no for SQL and PLpgSQL)
- Relational Database Design: Functional Dependency, Normal Forms, Design Algorithms for 3rd normal form and BC normal form (3.5 normal form)

Data models

- Data models: ER, Relational Data Model and their mapping

Entity-Relationship Model(cont)

Entity type: Group of object with the same properties

Entity: member of an entity type - analogous to an object.

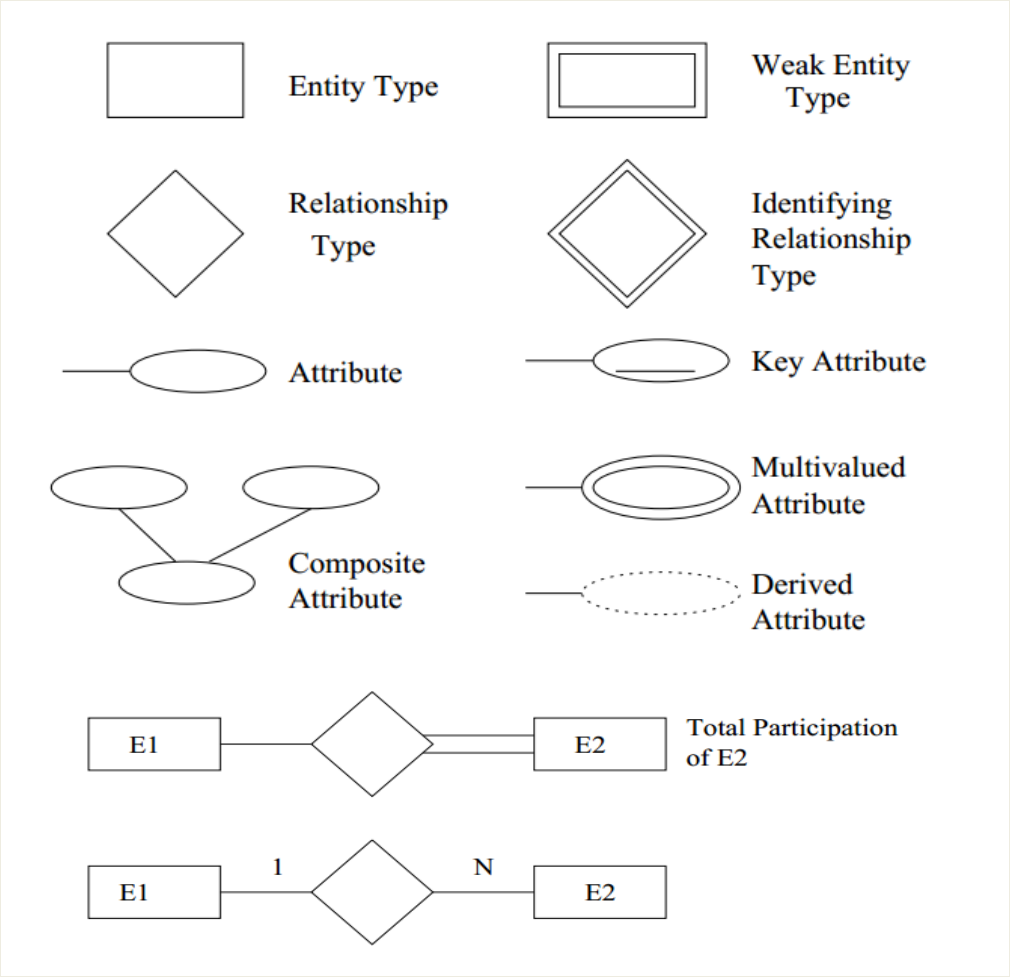
Attribute: a property of object

Relationship: among objects

- ER can model “n-way” relationship,
- ER models a relationship and its inverse by a single relationship.

Attributes of relationship types(cont)

The notation used for ERDs is summarised in Elmasre/Navathe Figure 3.15.



Relational Data Model

In the relational model, everything is described using relations.

A relation can be thought of as a named table.

Each column of the table corresponds to a named attribute.

The set of allowed values for an attribute is called its domain.

Each row of the table is called a tuple of the relation.

N.B. There is no ordering of column or rows.

Keys

Keys are used to identify tuples in a relation.

A *superkey* is a set of attributes that uniquely determines a tuple.

Note that this is a property of the relation that does not depend on the current relation instance.

A *candidate key* is a superkey, none of whose *proper* subsets is a superkey.

Keys are determined by the applications.

Integrity constraints

There are several kinds of integrity constraints that are an integral part of the relational model:

Key constraint: candidate key values must be unique for every relation instance.

Entity integrity: an attribute that is part of a primary key cannot be NULL.

Referential integrity: The third kind has to do with “foreign keys”.

Foreign Keys

Foreign keys are used to refer to a tuple in another relation.

A set, *FK*, of attributes from a relation schema R1 may be a foreign *key* if

- the attributes have the same domains as the attributes in the primary key of another relation schema R_2, and
- a value of FK in a tuple t_1 of R_1 either occurs as a value of PK for some tuple t_2 in R_2 or is null.

Referential integrity: The value of *FK* must occur in the other relation or be entirely NULL.

Relational Data Model vs ER Model

Relation schema (intension) ⇄ entity or relationship type schema (intension).

attributes ⇄ attributes

tuple ⇄ instance of entity/relationship

relation (instance, extension) ⇄ entity/relationship extension

composite and multivalued attributes are allowed in ER model, but not allowed in relational data model.

ER to Relational Data Model Mapping

One technique for database design is to first design a conceptual schema using a high-level data model, and then map it to a conceptual schema in the DBMS data model for the chosen DBMS.

Here we look at a way to do this mapping from the ER to the relational data model.

It involves 7 steps (see details in the lecture notes of Relational Data Model).

Relational Algebra

- Relational Algebra: be able to use relational algebra to answer question.

Relational Algebra is a procedural data manipulation language (**DML**).

It specifies operations on relations to define new relations:

Unary Relational Operations: Select, Project

Operations from Set Theory: Union, Intersection, Difference, Cartesian Product

Binary Relational Operations: Join, Divide.

OPERATION	PURPOSE	NOTATION
SELECT	Selects all tuples that satisfy the selection condition from a relation R	$\sigma_{<selection\ condition>}(R)$
PROJECT	Produces a new relation with only some of the attributes of R, and removes duplicate tuples.	$\pi_{<attribute\ list>}(R)$
THETA-JOIN	Produces all combinations of tuples from R and S that satisfy the join condition.	$R \bowtie_{<join\ condition>} S$
EQUI-JOIN	Produces all the combinations of tuples from R and S that satisfy a join condition with only equality comparisons.	$R \bowtie_{<join\ condition>} S$
NATURAL-JOIN	Same as EQUIJOIN except that the join attributes of S are not included in the resulting relation; if the join attributes have the same names, they do not have to be specified at all.	$R \bowtie_{<join\ condition>} S$
UNION	Produces a relation that includes all the tuples in R or S or both R and S; R and S must be union compatible.	$R \cup S$
INTERAECTION	Produces a relation that includes all the tuples in both R and S; R and S must be union compatible.	$R \cap S$
DIFFERENCE	Produces a relation that includes all the tuples in R that are not in S; R and S must be union compatible.	R - S
CARTESIAN PRODUCT	Produces a relation that has the attributes of R and S and includes as tuples all possible combinations of tuples from R and S.	$R \times S$
DIVISION	Produces a relation T(X) that includes all tuples t[X] in R(Z) that appear in R in combination with every tuple from S(Y), where Z = X $\cup$ Y.	$R(Z) \div S(Y)$

Database Languages

- Database Languages: SQL, PLpgSQL

(final exam: need be able to determine yes or no for SQL and PLpgSQL)

SQL Queries(cont)

Query syntax is:

SELECT attributes

FROM relations

WHERE condition

The result of this statement is a table, which is typically displayed on output.

The SELECT statement contains the functionality of *select*, *project* and *join* from the relational algebra.

SQL Identifiers

Names are used to identify objects such as tables, attributes, views, ...

Identifiers in SQL use similar conventions to common programming languages:

- a sequence of alpha-numerics, starting with an alphabetic,
- not case-sensitive,
- reserve word disallowed, ...

SQL Keywords

Some of the frequently-used ones:

- ALTER AND CREATE
- FROM INSERT NOT OR
- SELECT TABLE WHERE

For PostgreSQL Keywords see the Appendex of PostgreSQL doc .

SQL Data Types

All attributes in SQL relations have domain specified.

SQL supports a small set of useful built-in data types: strings, numbers, dates, bit-strings.

Self defined data type is allowed in PostgreSQL.

Various type conversions are available:

- date to string, string to date, integer to real ...
- applied automatically “where they make sense”

SQL Data Types(cont.)

Basic domain (type) checking is performed automatically.

Constraints can be used to “enforce” more complex domain membership conditions.

The NULL value is a member of all data types.

SQL Data Types(cont.)

Comparison operators are defined on all types.

< > <= >= = !=

Boolean operators AND, OR, NOT are available within WHERE expressions to combine results of comparisons.

Comparison against NULL yields FALSE.

Can explicitly test for NULL using:

- ***attr* IS NULL** ***attr* IS NOT NULL**

Most data types also have type-specific operations available (e.g. arithmetic for numbers).

Which operations are actually applied depends on the implementation.

Relational Database Design

- Relational Database Design: Functional Dependency, Normal Forms, Design Algorithms for 3rd normal form and B-C normal form (3.5 normal form)

Functional dependencies

A function *f* from S_1 to S_2 has the property

$$if\ x, y\ \in S_1 \text{ and } x = y, \text{then}\ \ f(x) = f(y)\ .$$

A generalization of keys to avoid design flaws violating the above rule.

Let X and Y be sets of attributes in R.

X (*functionally*) determines *Y* , $X \rightarrow Y$, iff $t_1[X] = t_2[X]$ implies $t_1[Y] = t_2[Y]$.

i.e., f (t(X)) = t [Y]

We also say $X \rightarrow Y$ is a *functional* dependency, and that Y is *functionally* dependent on X.

X is called the *left side*, Y the *right side* of the dependency.

Armstrong's axioms (1974)

Notation: If X and Y are sets of attributes, we write XY for their union.

e.g. X = {A, B}, Y = {B, C}, XY = {A, B, C}

F1 (Reflexivity) If $X \supseteq Y$ then $X \rightarrow Y$.

F2 (Augmentation) $\{X \rightarrow Y\} \models XZ \rightarrow YZ$.

F3 (Transitivity) $\{X \rightarrow Y, Y \rightarrow Z\} \models X \rightarrow Z$.

F4 (Additivity) $\{X \rightarrow Y, X \rightarrow Z\} \models X \rightarrow YZ$.

F5 (Projectivity) $\{X \rightarrow YZ\} \models X \rightarrow Y$.

F6 (Pseudotransitivity) $\{X \rightarrow Y, YZ \rightarrow W\} \models XZ \rightarrow W$.

Algorithm to compute X^+

$X^+ := X$;

change := true;

while change do

begin

change := false;

for each FD $W \rightarrow Z$ in F do

begin

if ($W \subseteq X^+$) and ($Z \not\subseteq X^+$) then do

begin

$X^+ := X^+ \cup Z$;

change := true;

end

end

end

Algorithm to Compute a Candidate Key

Given a relational schema *R* and a set *F* of functional dependencies on R.

A key *X* of *R* must have the property that $X^+ = R$.

Algorithm to compute a candidate key

Step 1: Assign *X* a superkey in F.

Step 2: Iteratively remove attributes from *X* while retaining the property $X^+ = R$ till no reduction on *X*.

The remaining *X* is a key.

Algorithm to Compute All the Candidate Keys

Given a relational schema R and a set F of functional dependencies on R, the algorithm to compute all the candidate keys is as follows:

T := ∅

Main:

X := S where S is a super key which does not contain any candidate key in T

remove := true

While remove do

For each attribute A ∈ X

Compute $\{X-A\}^+$ with respect to F

If $\{X-A\}^+$ contains all attributes of R then

X := X – {A}

Else

remove := false

T :=T ∪ X

Repeat *Main* until no available S can be found. Finally, T contains all the candidate keys.

Normal Forms for Relational Databases

Normal Forms:

- 1NF, 2NF, 3NF (Codd 1972)
- Boyce-Codd NF (1974)
- Multivalued dependencies and 4NF (Zaniolo 1976 and Fagin 1977)
- Join dependencies (Rissanen 1977) and 5NF (Fagin 1979)

First Normal Form (1NF)

This simply means that attribute values are *atomic*, and is part of the definition of the relational model.

Atomic: multivalued attributes, composite attributes, and their combinations are disallowed.

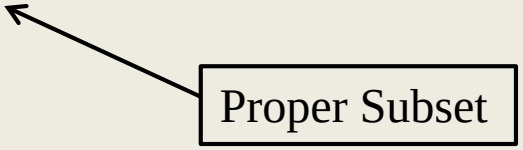
There is currently a lot of interests in non-first normal form databases, particularly those where an attribute value can be a table (nested relations).

Consider the table below, adapted from Desai.

Second Normal Form (2NF)

A *prime* attribute is one that is part of a candidate key. Other attributes are *non-prime*.

Definition: In an FD $X \rightarrow Y$, Y is *fully functionally dependent* on X if there is no $Z \subset X$ such that $Z \rightarrow Y$. Otherwise Y is *partially* dependent on X.



Definition (*Second Normal Form*): A relation scheme is in second normal form (2NF) if all non-prime attributes are fully functionally dependent on the relation keys.

A database scheme is in 2NF if all its relations are in 2NF.

Third Normal Form (3NF) (cont)

Definition (Third Normal Form): A relation scheme is in *third normal form (3NF)* if for all non-trivial FD's of the form X→A that hold, either X is a superkey or A is a prime attribute.

Note: a FD X → Y is trivial iff Y is a subset of X.

Alternative definition: A relation scheme is in third normal form if every non-prime attribute is fully functionally dependent on the keys and not transitively dependent on any key.

A database scheme is in 3NF if all its relations are in 3NF.

Boyce-Codd Normal Form (BCNF)

Definition (*Boyce-Codd Normal Form*):

> A relation scheme is in *Boyce-Codd* Normal Form (BCNF) if whenever
>
> $X \rightarrow A$ holds and $X \rightarrow A$ is non-trivial, X is a superkey.

A database scheme is in BCNF if all its relations are in BCNF.

We can make our example into BCNF:

Relational Database Design

Anomalies can be removed from relation designs by decomposing them until they are in a normal form.

Several problems should be investigated regarding a decomposition.

A decomposition of a relation scheme, R, is a set of relation schemes $\{R_1, \ldots, R_n\}$ such that $R_i \subseteq R$ for each i, and $\bigcup_{i=1}^{n} R_i = R$

Note that in a decomposition $\{R_1, \ldots, R_n\}$, the intersect of each pair of R_i and R_j does not have to be empty.

Example: R = {A, B, C, D, E}, R_1 = { A, B}, R_2 ={A, C}, R_3 = {C, D, E}

A naive decomposition: each relation has only attribute.

A good decomposition should have the following two properties.

Dependency Preserving

Definition: Two sets F and G of FD's are equivalent if $F^+ = G^+$.

Given a decomposition $\{R_1, \ldots, R_n\}$ of R:

$$F_i = \{X \to Y : X \to Y \in F \& X \in R_i, Y \in R_i\}.$$

The decomposition $\{R_1, \ldots, R_n\}$ of R is dependency preserving with respect to F if

$$F^+ = \left(\cup_{i=1}^{i=n} F_i\right)^+.$$

Lossless Join Decomposition

A second necessary property for decomposition:

A decomposition $\{R_1, \ldots, R_n\}$ of R is a *lossless join* decomposition with respect to a set F of FD's if for every relation instance r that satisfies F:

$$r = \pi R_1(r) \bowtie \cdots \bowtie \pi R_n(r).$$

If $r \subset \pi R_1(r) \bowtie \cdots \bowtie \pi R_n(r)$, the decomposition is *lossy.*

Lossless decomposition into BCNF

Algorithm TO_BCNF

D := $\{R_1, R_2, \ldots R_n\}$

While ∃ a $R_i \in$ D and R_i is not in BCNF **Do**

{ find a X →Y in R_i that violates BCNF; replace R_i in D by (R_i − Y) and (X ∪ Y); }

Since a X →Y violating BCNF is not always in F, the main difficulty is to verify if R_i is in BCNF; see the approach below:

1. For each subset X of R_i, computer X^+.
2. $X \rightarrow (X^+|_{Ri} - X)$ violates BCNF, if $X^+|_{Ri} - X \neq \emptyset$ and $R_i - X^+ \neq \emptyset$.

Here, $X^+|_{Ri} - X = \emptyset$ means that each F.D with X as the left hand side is trivial;

$R_i - X^+ = \emptyset$ means X is a superkey of R_i

Lossless and dependency-preserving decomposition into 3NF

A lossless and dependency-preserving decomposition into 3NF is always possible.

More definitions regarding FD's are needed.

A set *F* of FD's is minimal if

1. Every FD $X \rightarrow Y$ in F is simple: Y consists of a single attribute,
2. Every FD $X \rightarrow A$ in F is *left-reduced*: there is no proper subset

   $Y \subset X$ such that $X \rightarrow A$ can be replaced with $Y \rightarrow A$.

   that is, there is no $Y \subset X$ such that

   Iff F |= Y → A

$$((F - \{X \rightarrow A\}) \cup \{Y \rightarrow A\})^+ = F^+$$

3. No FD in F can be removed; that is, there is no FD $X \rightarrow A$ in F

   such that

   Iff X→A is inferred From F– { X→A}

$$(F - \{X \rightarrow A\})^+ = F^+.$$

Computing a minimum cover

F is a set of FD's.

A *minimal cover* (or *canonical cover*) for F is a minimal set of FD's F_{min} such that $F^+ = F^+_{min}$.

Algorithm Min_Cover

Input: a set F of functional dependencies.

Output: a minimum cover of F.

Step 1: *Reduce right side*. Apply Algorithm Reduce right to F.

Step 2: *Reduce left side*. Apply Algorithm Reduce left to the output of Step 2.

Step 3: *Remove redundant* FDs. Apply Algorithm Remove_redundency to the output of Step 2. The

output is a minimum cover.

Below we detail the three Steps.

Computing a minimum cover(cont)

Algorithm Reduce_right

INPUT: F.

OUTPUT: right side reduced F'.

For each FD $X \rightarrow Y \in F$ where Y = $\{A_1, A_2, ..., A_k\}$, we use all $X \rightarrow \{A_i\}$ (for $1 \leq i \leq k$) to replace $X \rightarrow Y$.

Algorithm Reduce_left

INPUT: right side reduced F.

OUTPUT: right and left side reduced F'.

For each $X \rightarrow \{A\} \in F$ where X = $\{A_i : 1 \leq i \leq k\}$, do the following. For i = 1 to k, replace X with $X - \{A_i\}$ if A $\in (X - \{A_i\})^+$.

Algorithm Reduce_redundancy

INPUT: right and left side reduced F.

OUTPUT: a minimum cover F' of F.

For each FD $X \rightarrow \{A\} \in F$, remove it from F if: $A \in X^+$ with respect to $F - \{X \rightarrow \{A\}\}$.

Overview: DBMS

- Disk, Files, Buffer Replacement Policy
- Indexing Basic, Hash-based Index, Tree-Structed Index
- Transaction Management
  - ACID properties
  - Various schedules: Serializable, Conflict-Serializable, Schedule Graph, Wait for Graph, …
  - concurrency control (locking, time-stamp ordering) --- for multi-versioning, optimistic, only need to know the basic idea.
- Graph Processing. (Not examined)
- Graph Systems (Not examined)

Buffer Replacement Policies

- Least Recently Used (LRU)
  - release the frame that has not been used for the longest period.
  - intuitively appealing idea but can perform badly
- First in First Out (FIFO)
  - need to maintain a queue of frames
  - enter tail of queue when read in
- Most Recently Used (MRU):
  - release the frame used most recently
- Random

No one is guaranteed to be better than the others. Quite dependent on applications.

Files

A ***file*** consists of several data blocks.

Heap Files: unordered pages (blocks).

Two alternatives to maintain the block information:

- Linked list of pages.
- Directory of pages.

Three types of file organisations

- Heap, Sorted and Hashed

The COST of processing DB queries

- SCAN, Search (Single, Range), Insert, Delete

Summary

File Type	Scan	Equality Search	Range Search	Insert	Delete
Heap	BD	0.5 BD	BD	Search + D	Search + D
Sorted	BD	D logB	D log B + # matches	Search + BD	Search + BD
Hashed	1.25 BD	D	1.25 BD	2 D	Search + BD

A Comparison of I/O Costs

(I/O Costs: The cost of reading/writing from/to disk)

Indexing Structure

- Index is collection of data entries k∗.
  - Each data entry k∗ contains enough information to retrieve (one or more) records with search key value k.

- Unclustered VS Clustered Index

- Dense VS Sparse Index

- Primary and Secondary Index

Indexing Techniques

As for any index, 3 alternatives for data entries **k***:

- 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 is orthogonal to the *indexing technique* used to locate data entries **k***.

- Tree-Structured indexes are best for **sorted access** and **range queries**.
  - ***ISAM*:** static structure;
  - ***B+ tree*:** dynamic, adjusts gracefully under inserts and deletes.
- Hash-based indexes are best for **equality selections**. **Cannot** support range searches.
  - Static and dynamic hashing techniques exist.

ISAM

- *File creation*: Leaf (data) pages allocated sequentially, sorted by each key; then index pages allocated, then space for overflow pages.
- ***Index entries*: <search key value, page id>**; they **'direct' search for *data entries***, which are in leaf pages.
- *Search*: Start at root; use key comparisons to go to leaf. Cost $\log_F N$;

  F = # entries/index pg, N = # leaf pgs

- *Insert*: Find leaf data entry belongs to, and put it there if space is available, else allocate an overflow page, put it there, and link it in.
- *Delete*: Find and remove from leaf; if empty overflow page, de-allocate.

Static tree structure: *inserts/deletes affect only leaf pages.*

Inserting a Data Entry into a B+ Tree

1. Find correct leaf *L*.
2. Put data entry onto *L*.
   - If *L* has enough space, *done*!
   - Else, must ***split*** *L (into L and a new node L2)*
     - ➢Redistribute entries evenly, **copy up** middle key.
     - ➢Insert index entry pointing to *L2* into parent of *L*.
3. This can happen recursively
   - **To split index node**, redistribute entries evenly, but **push up** middle key. (Contrast with leaf splits.)
4. Splits ‘grow’ tree; root split increases height.
   - Tree growth: gets ***wider*** or ***one level taller at top.***

Deleting a Data Entry from a B+ Tree

1. Start at root, find leaf *L* where entry belongs.
2. Remove the entry.
   - If L is at least half-full, *done!*
   - If L has only $\lceil (\mathbf{p}/2) \rceil$ **-1** entries,
     - Try to **re-distribute**, borrowing from *sibling (adjacent node with same parent as L)*.
     - If re-distribution fails, ***merge*** *L* and sibling.
3. If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
4. Merge could propagate to root, decreasing height.

Dynamic Hashing Example

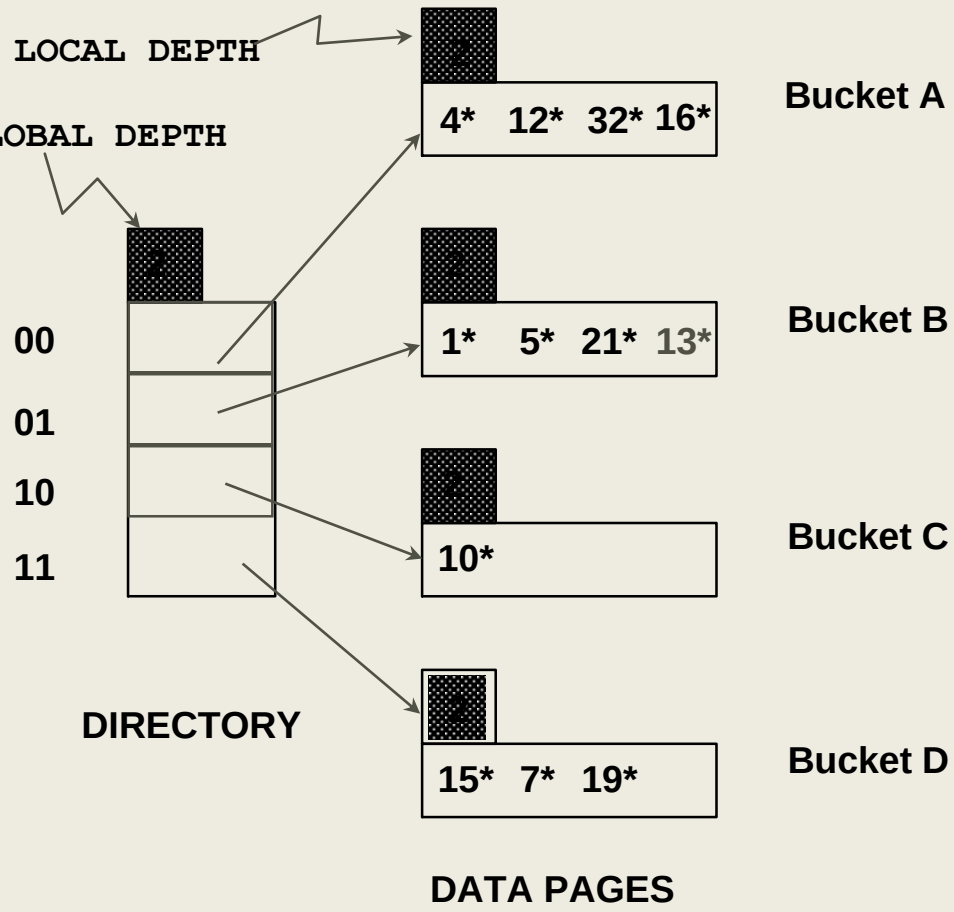
Directory is array of size 4.

To find bucket for *r*, take last `***global depth***' # bits of **h**(*r*); we denote *r* by **h**(*r*).

- If **h**(*r*) = 5 = binary 101, it is in bucket pointed to by 01.

❖ **Insert**: If bucket is full, ***split*** it (*allocate new page, re-distribute*).

❖ *If necessary*, double the directory.
(As we will see, splitting a bucket does not always require doubling; we can tell by comparing ***global depth*** with ***local depth*** for the split bucket.)



Linear Hashing

- **Splitting proceeds in `rounds'.** Round ends when all N_R initial (for round *R*) buckets are split. Buckets 0 to *Next-1* have been split; *Next* to N_R yet to be split.
- **Current round number is *Level*.**
- **Search:** To find bucket for data entry *r*, find $\mathbf{h}_{Level}(r)$:
  - If $\mathbf{h}_{Level}(r)$ in range `*Next* to N_R', *r* belongs here.
  - Else, r could belong to bucket $\mathbf{h}_{Level}(r)$ or bucket $\mathbf{h}_{Level}(r) + N_R$; must apply $\mathbf{h}_{Level+1}(r)$ to find out.
- **Insert:** Find bucket by applying $\mathbf{h}_{Level}$ / $\mathbf{h}_{Level+1}$:
  If bucket to insert into is full:
  - Add overflow page and insert data entry.
  - (*Maybe*) Split *Next* bucket and increment *Next*.

Desirable Properties of Transaction Processing **ACID**

- ***Atomicity***: A transaction is either performed in its entirety or not performed at all.
- ***Consistency preservation***: A correct execution of the transaction must take the database from one consistent state to another.
- ***Isolation***: A transaction should not make its updates visible to other transactions until it is committed.
- ***Durability or permanency***: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

Check Conflict Serializability

Algorithm

Step 1: Construct a *schedule* (or *precedence*) graph – a *directed graph*.

Step 2: Check if the graph is *cyclic*:

- Cyclic: non-serializable.
- Acyclic: serializable.

Construct a Schedule Graph $G_S = (V, A)$ for a schedule S

1. A vertex in V represents a transaction.
2. For two vertices T_i and T_j, an arc $T_i \rightarrow T_j$ is added to A if
   - there are two *conflicting* operations $O_1 \in T_i$ and $O_2 \in T_j$,
   - in S, O_1 is before O_2.

Two operations O_1 and O_2 are *conflicting* if

- they are in different transactions but on the same data item,
- one of them must be a write.

Locking Rules

In this schema, every transaction *T* must obey the following rules.

1) If *T* has only one operation (read/write) manipulating an item *X*:

- obtain a read lock on *X* before reading it,
- obtain a write lock on *X* before writing it,
- unlock *X* when done with it.

2) If *T* has several operations manipulating *X*:

- obtain one proper lock only on *X*:

  a read lock if all operations on *X* are reads;

  a write lock if one of these operations on *X* is a write.
- unlock *X* after the last operation on *X* in *T* has been executed.

Timestamp ordering

The idea here is:

- to assign each transaction a timestamp (e.g. start time of transaction), and
- to ensure that the schedule used is equivalent to executing the transactions in timestamp order

Each data item, X, is assigned

- a read timestamp, *read TS*(*X*) - the latest timestamp of a transaction that read X, and
- a write timestamp, *write TS*(*X*) - the latest timestamp of a transaction that write X.

Final Exam

- 2 hrs
- Based on understanding
- **If you do not feel well on the exam day, please not attend the exam. If you attend the exam, no sup-exam will be given!**
- **Consultation: One week prior to the final exam.**
- **Sample questions will be out soon. Please note that sample questions just reflect the difficult degree but not the scope nor the similarity.**