**CS2855 Databases**

Reading = Database System Concepts, 6th edition, Avi Silberschatz

Coursework:

* 3 coursework assignments (15%)
* Mini-project (25%)
* Exam (60%)

Text -> ER Diagram -> Relational Model -> Relational Algebra -> SQL

* DBMS:
  + Database (collection of interrelated data)
  + programs to access them
  + env = convenient + efficient
* Database Application:
  + banking, airlines....
* Database = fundamental to ~all systems

Purpose:

* Drawbacks of file:
  + **Data redundancy + inconsistency**
  + **difficult in accessing**
  + **atomicity of updates**
    - failure = inconsistent = partial updates
  + **concurrent access**
  + **security (constraints)**
  + **No data independence**

Level of "Abstraction"

1. Physical (how it's stored)
2. Logical (how data (type) stored in database, relationship between data)
3. View (app hide details of data type, hide info for security)

Data Model

* Data
* Relationship
* semantics
* constraints

1. ER Data Model for DB Design
2. Relational Model
3. Semistructured Data Model (XML)

Data Schemas VS Instances

* schema = overall logical design of DB
* instance = snapshot of the DB, the actual data in the DB at that time
* instances changes, schema almost never do

ER Model

* Model = entities + relationships
  + entity = thing or object (distinguishable from other objects)
  + attributes = entities
  + relationship = association among several entities

Relational Model

* Model = collection of tables
* lower level than ER Model
* ER Design -> Relational model

Data Definition Language (DDL)

* defines schema
* DDL compiler = generates set of tables (into data dictionary)
* Data Dictionary = contains metadata (schema, storage structure, access, integrity constraints, authorization)

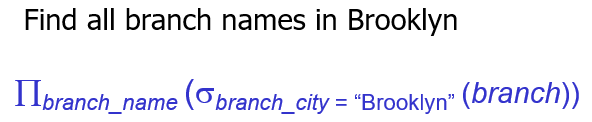
Data Manipulation Language (DML)

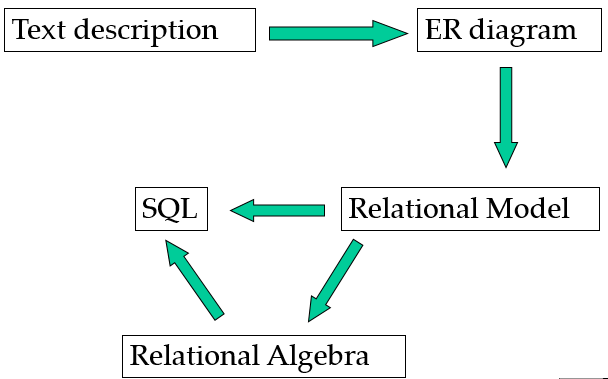
* Access + manipulate
* query + update
* Procedural (what data, how)
* Declarative (what data is required)
* “Pure languages” = relational algebra

Extensible Markup Language (XML)

* general purpose, user defines tag

Not separated (SQL will consist of both)

Relational Algebra

SQL (Standard SQL examples)

**Part 2: ER Model  
ER Model** = high level data model

* collection of entities
* relationships
* constraints
* attributes

**Entities: thing**

* have attributes
* entity set = set of entities of same type, same kind of attributes
* entity has value for each attributes

**Relationship: association among several entities**

* Relationship-set = set of tuples (relationships) Role: function that an entity plays in a R  
  E.g. **R** = Joan is father of Sean. **E** = Joan, Sean. **Role:** parent-child relationship. Can have 2 roles

**Descriptive attribute**: an attribute of a relationship set

**Degree** = no. of entity sets in relationship set

* binary = 2
* ternary = 3

**Attributes** = properties  
**Domain** = set of permitted values for each attr  
Attribute of an entity = a function that maps entity set onto a domain  
entity = set of pairs (attribute, data value)

**Types of Attributes**

* Simple, Composite attributes
* Single-valued, Multi-valued attributes
* Derived attributes (computed from other attr

**Constraints**

**M**apping cardinalities

* no of entities to which another entity can be associated via a relationship set
* binary = 1 to 1, 1 to \*, \* to 1, \* to \*

**Ternary Relationships**

* only 1 arrow out of a ternary relationship
* if > 1 = confusing

A ternary relationship *R* between *A*, *B* and *C* with arrows to *B* and *C* could mean  
1. Each *A* entity is associated with a unique entity from both *B* and *C ; or*  
2. Each pair of entities from (*A, B*) is associated with a unique *C* entity, and each pair (A, C) is associated with a unique B

**Keys**

* Differences= in the attributes
* how entities & relationships are distinguished
* key = allow us to distinguish them
* **Super Key** = set of one/more attr sufficient to uniquely determine an entity
* **Candidate key** = minimal super key
* **Primary key** = candidate key chosen
* **Foreign Key** = an attr corresponds to the primary key of another relation scheme

\*Super Key for relationships =   
PrimaryKey(E1) U PrimaryKey(E2)  
**Participation constraints**

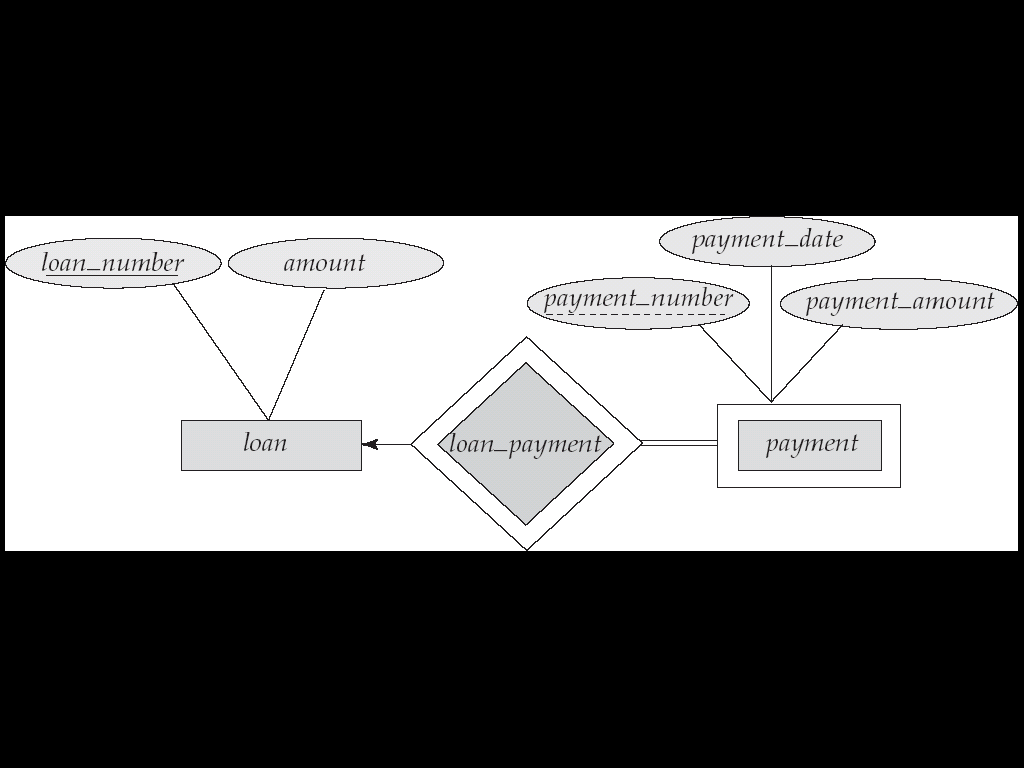
* Participation of an entity set = total (if every entity in E participates in at least one relationship in R)
* or else = partial

**Roles**

* labelling lines that connects diamonds to rectangle
* (optional) clarification (e.g, manager, worker)

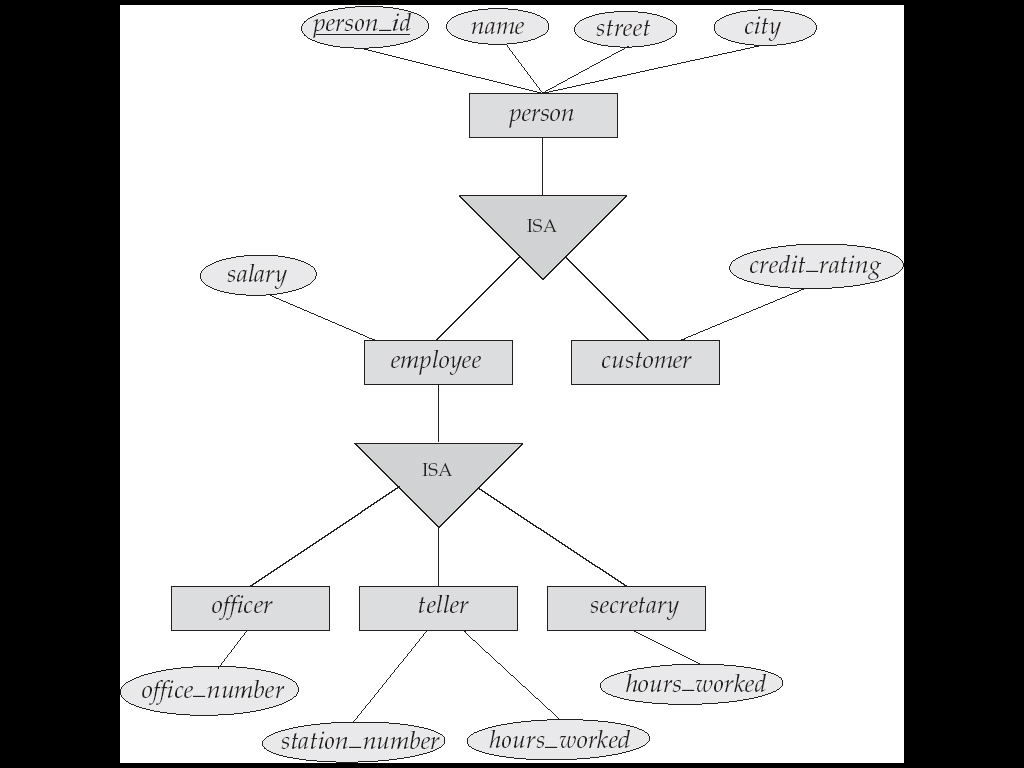
**Design Issues**

* Use of entity sets VS attributes (enterprise)
* Use of entity sets VS relationship sets (use R, if an action occurs between entities)
* Binary VS n-nary relationship sets (if n-nary is more clear, than several binary, partial info)
* Placement of relationship attributes
* Mapping Cardinalities (access-date can be attribute of acc if each account can have only one customer)  
  **Weak Entity Sets**
* no primary key
* existence = depends on existence of identifying entity set
* **discriminator/partial key** = super key of a weak entity set
* primary key = (primary K of strong set) + partial K
* more appropriate as an attribute if participates in only identifying relationship, and has few attribute (multivalued composite attribute)



**Specialization**

* attribute inheritance (lower level inherits attribute + relationship of higher level)

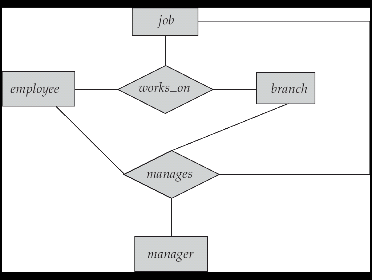
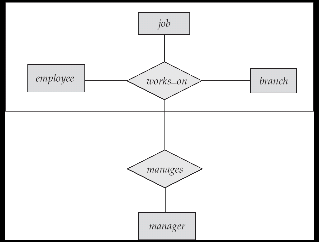


**Generalization**

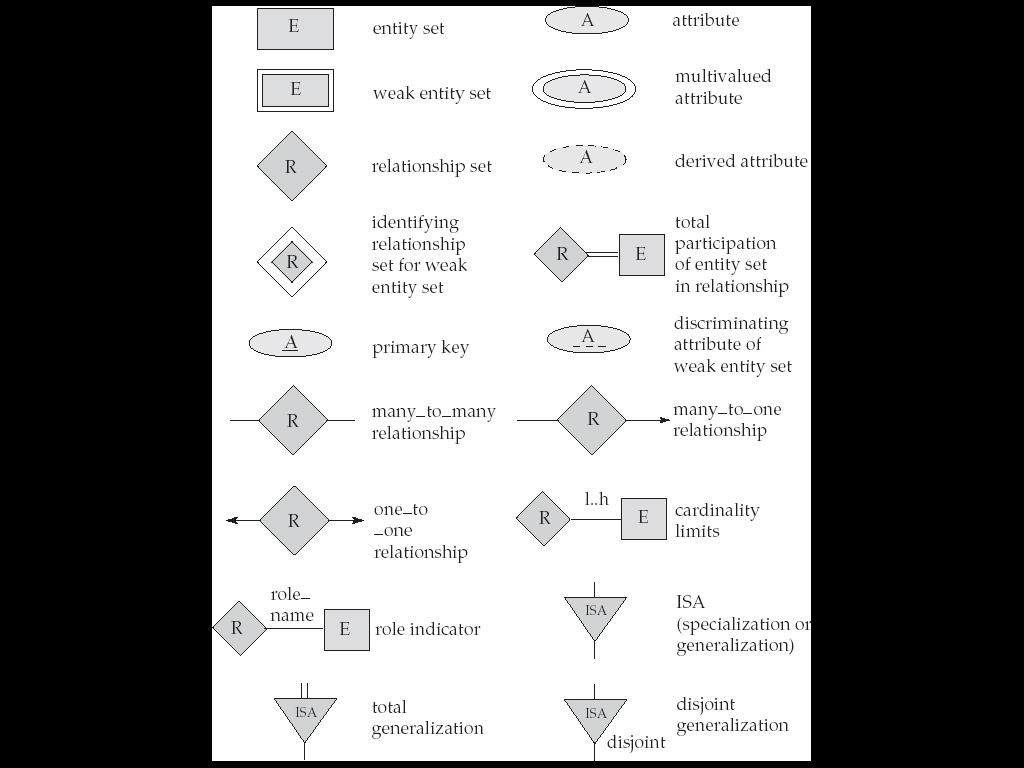
Opposite of Specialization

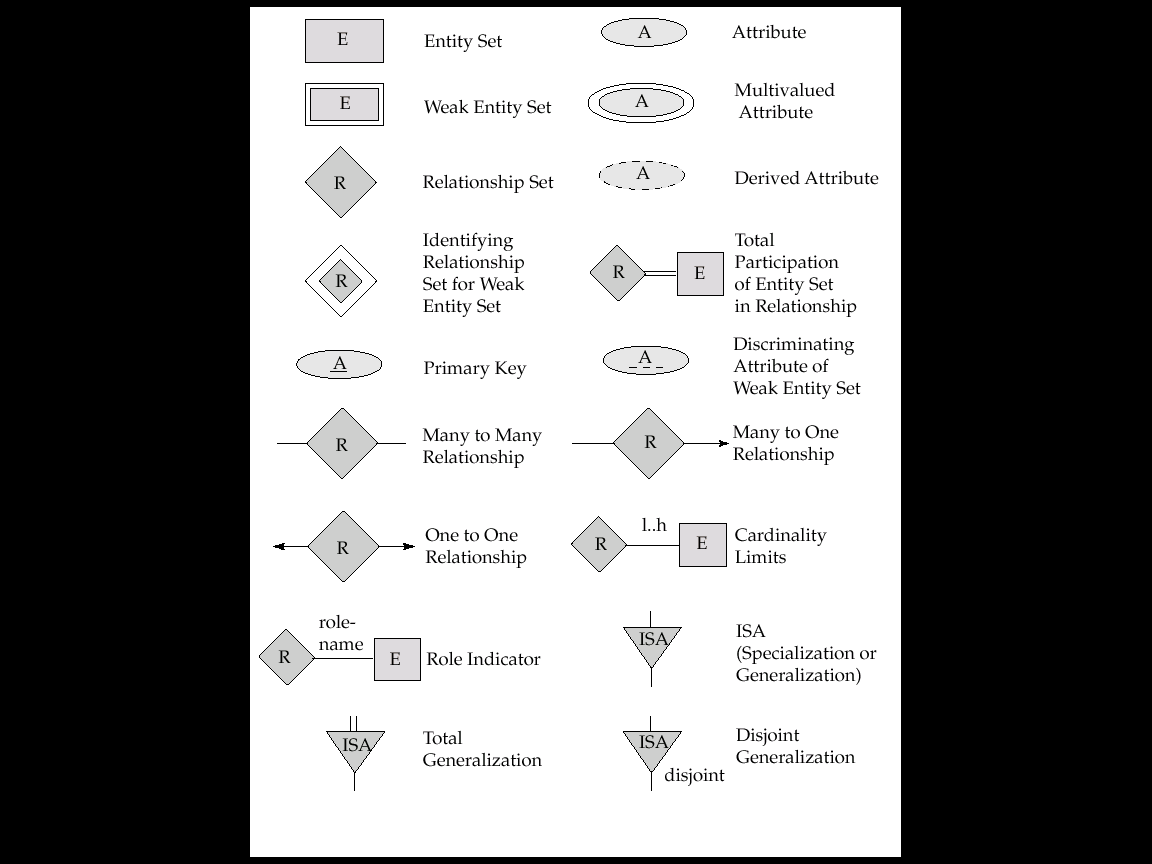
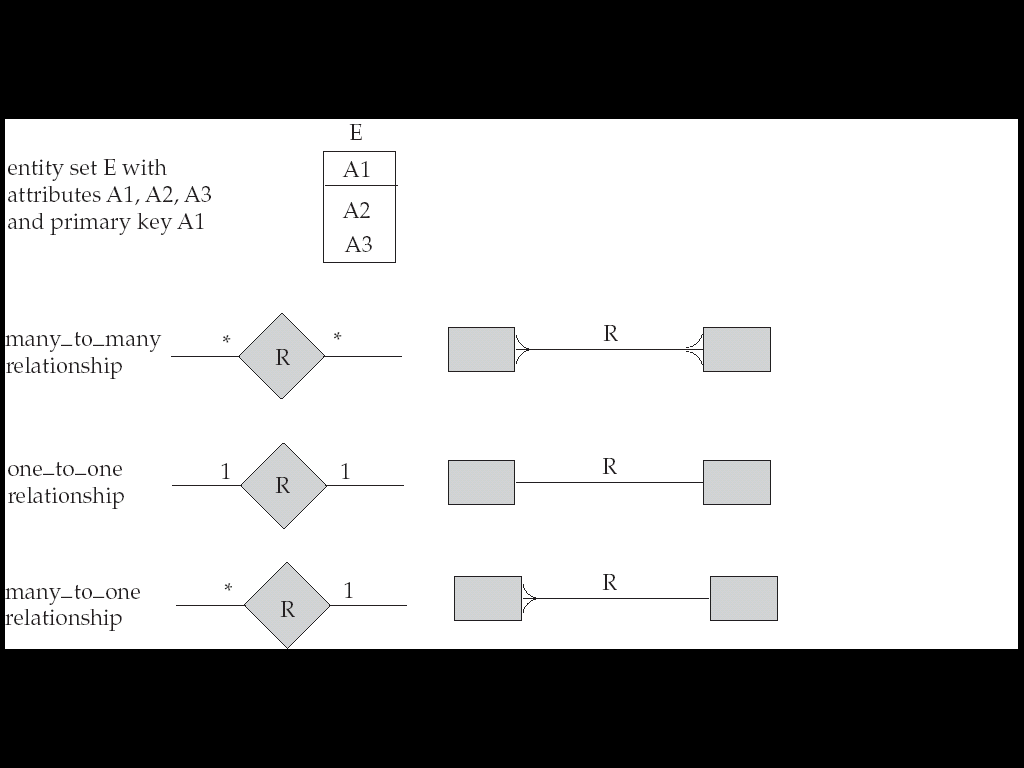
* + - 1. Constraints be members of lower level
      2. Constraints on belong to more than one lower level
         1. Disjoint – 1 lower-level
         2. Overlapping – more than 1 lower-level
      3. Completeness Constraint
         1. Total = must belong to one of the lower level
         2. Partial = need not belong to…

**Aggregation**

* eliminate redundancy
* treat relationship as an abstract new entity
* allow relationships between relationship 

**Shapes**



**Relational Model**

Conceptual Design of Database

* Relation
* Relation Schema
* Relation instance
* Key
* Schema Diagrams

Transform ERD to Relational Model

Relational Algebra = specify request for info

**Structure of Relational Model**

Consists of collection of tables (unique name)

Row in table = relationship among set of value

Headers = attributes

Permitted values = domain

Relation *R*  is a subset of D1 x D2 x D3

Relation is a set of n-tupes (a1, a2, .. , an) where ai E Di

**Attribute**

Each attribute of a relation has name

Allowed values of attribute = domain of attribute

Should be atomic (indivisible, cannot be broken down)

\*Therefore domain is atomic if members are atomic

Special value *NULL* = member of every domain

**Relation Schema** = list of attributes \*like type

Cus\_schema= (cus\_name, cus\_street, cus\_city) \*like var

**Relation**  on the relation schema **R =** r(R)

Customer(cus\_schema)

**Relation instance** = current value \*like value of var

Element t of r is a tuple(row)

**Order of tuples** = irrelevant

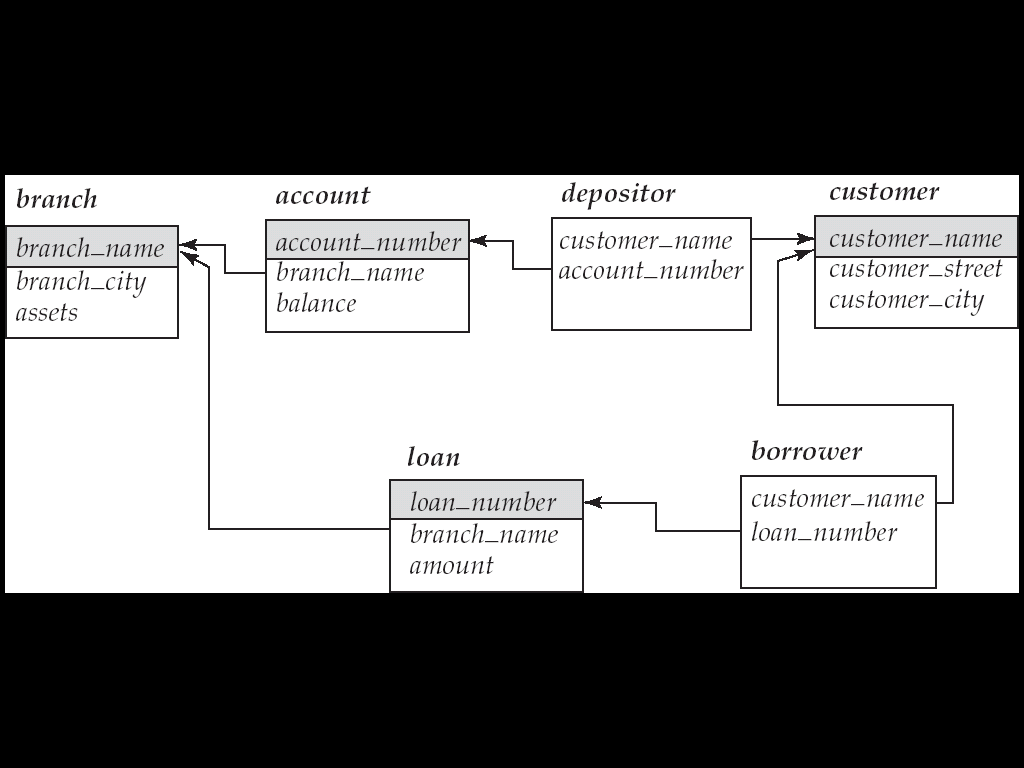
**Normalization Theory**

Info = broken into parts, each relation storing one part

Store all info into one single relation = repetition of info

**Foreign Keys**

**Convert ERD to relational Schemas**



Database = can be represented by collection of schemas

Each Entity and Relationship = relation schema

Constraints (From ERD) mapped to relation schemas

**Representation of Strong Entity Set**

E with n distinct attributes (PK is the same PK)  
*E.g. loan = (loan\_number, amount )*  
**Representation of Weak Entity Set**

E = {a1, a2, am} U { b1, b2, bn}

PK = PK of strong + discriminator of weak

FK = PK of strong

*E.g. loan = (loan\_number, payment\_number, payment\_date, payment\_amount )*

**Representation of relationship set**

R = {a1, a2, am} U { b1, b2, bn}

a and b = descriptive for relation  
R formed by union of primary of both sets

\*..\* = PKa U PKb

1..1 = PKa OR PKb

1..\* = PK\*

\*..\*..\* = PKa U PKb U PKc

1..\*..\* = PK\*

**Combination of Schemas**

If TOTAL participation of A, then can combine A to AB into 1 schema

If 1..1, then can combine also

**Redundancy of Schemas**

R for weak to strong = no need to be present

**Composite Attributes**

Sub-Attr = flattened out, no separate attr for composite

**Multi-valued Attribute**

Create new schema for the MV Attribute with PK of the E  
MV Attr = (PKE, MV\_value)

**Representing Specialization via Schema**

**Method 1:***person = (person\_id, street, city)   
customer = (person\_id, credit\_rating)  
employee = (person\_id, salary)*

**Method 2: (**disjoint + total) *employee = (person\_id, name, street, city, salary)*

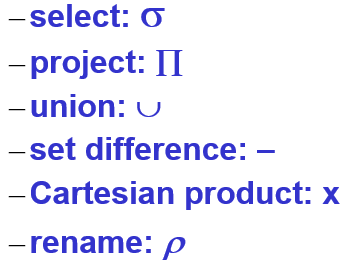
*customer = (person\_id, name, street, city, credit\_rating)*X = if there is R with the higher level entity set

**Relational Algebra**

QL = user request info from DB

* Procedural (Relational algebra)
* Non-procedural

6 basic operators

****

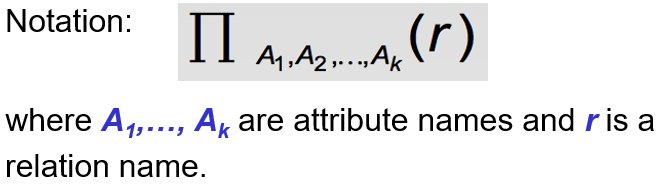
**SELECT**

****

p = formula which have:

* ^and, Vor, =, >, <

**PROJECT**



= show rows listed, and no duplicates

\*Duplicate rows are removed

**UNION**



R and S must be compatible:

* R, S same arity
* Attribute domains = compatible

**SET DIFFERENCE**



R and S must be compatible:

* R, S same arity
* Attribute domains = compatible

**CARTESIAN-PRODUCT**

Notation *r* x *s*

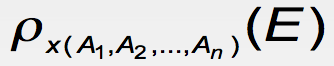
R and S = disjoint (if not, use dot borrower.name)

**COMPOSITION of Ops**

Can do nested operations ****

**RENAME Ops**

****



Name of attributes changed to A1, A2, An

**Set-Intersection Op**

****

R and S must be compatible:

* R, S same arity
* Attribute domains = compatible

**Natural-Join Op**

1. Forms a Cartesian product

2. Performs a selection, forcing equality between values  
3. Remove duplicate attributes

**Division Ops *r* ÷ *s***

Values of attribute A appear in r with every possible value of B in s

**Assignment Op 🡪**

Provides a convenient way to express complex queries

**Null**

Null = unknown or value does not exist

Null = null (in SQL)

Modification of the Database

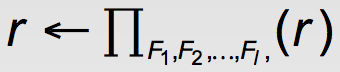
Deletion

* Selected tuples are removed
* Can only delete tuples (not values)
* r <- r - E

Insertion

* r <= r U E

Updating

* 

**SQL**

* Define structure of data
* Modify data
* Security constraints

**DDL and Interactive DML**

* Schema
* Domain
* Integrity constraints
* Security and authorization
* Physical storage structure

Data Definition

Domain Types

* Char, varchar, int, float

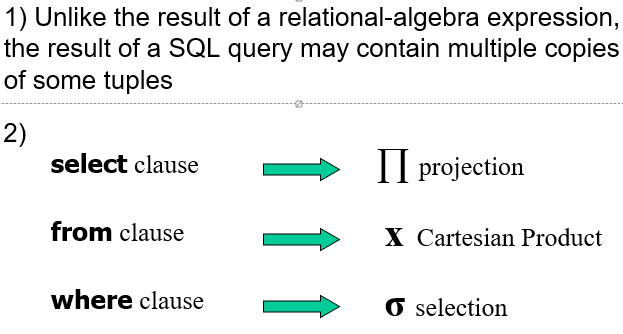
Create Table

* *Create table r (A1 D1,… (integrity cconstraint1))*

Drop/Alter Table

* Alter table r add A D

*Select (distinct) \* from loan as L where L.name=’mary’ and amount>1200 or L.name like ‘%hel&’ order by L.name asc*



Set Operation

* Auto remove duplicates
* *Union, intersect, except*

Aggregate functions

* *Avg, min, max, sum, count*
* Can use with *groupby, having*

Null Values

* *Is null, is not null*
* Ignores null values if using *count(\*)*

3 logic *= True/false/unknown*

Result of *where* is *false* (if *unknown*)

Set membership

* *In, not in*

Set comparison

* *Some, all*
* *Greater than at least one = >some*

Derived Relations

* Subquery expression used in the *from* clause
* 2 froms

With clause

* Temporary view
* Used to simply derived relations

Views

* Any relation not of the conceptual model made visible to user as virtual relation = view
* Use to hide certain data
* Create view v as <query>
* Stored the definition (not results)

View Expansion

* Not recursive, loop will terminate.

Database Modification

* Delete (will compute and select the ones to delete, and delete. Avg won’t be recomputed)
* *Insert*
* *Update* (order is important)

Case statement

* *Update account*

*Set balance = case*

*When balance <=10000*

*Then balance \*1.05*else balance \* 1.06

End

Update with Views

* Difficulty: a modification to db in terms of view  
  Therefore modifications not permitted on view
* Updatable : sql which modifications are allowed

Built in Data Types

* Date (year, month, date)
* Time (hours minute seconds)
* Timestamp (date plus time)
* Interval (period of time)

Can extract values from it (year from date)

Can cast string to date/time/timestamp

Allow comparisons

User-defined Types

* helpful for avoiding errors
* *Create type Dollars, as numeric (12,2) final*
  + *Final* cannot have subtype
* Cannot assign or compare (different types)
* Can cast
* Drop Type and Alter Type

User-defined Domains

* Most elementary form of integrity constraint
* Test value when inserted
* *create domain Dollars numeric (12,2)*

Large-Object Type

* blob: binary large object
* clob: character large object
* returns pointer to large object

Integrity Constraints

* Guard against accidental damage to db.
* Not null
* Primary key, foreign key
* Unique
* Check (P, where P is a predicate)
  + Check in the create domain
  + *create domain Dollars numeric (12,2)*

constraint value\_test check (value> = 4.00)

* can be added by using alter

alter table table\_name add constraint

Referential Integrity

* foreign key
* value must appear as a tuple in another table
* *foreign key (branch\_name) references branch*

Assertion

* a predicate, P, that db must always satisfy

*create assertion assertion\_name check predicate*

* checks on every update (might have great overhead)

Authorization

* Privileges
* Select, Insert, Update, Delete,
* Resources (new relations),   
  alteration (add/rm attributes),   
  drop (delete relations) <- schema
* All privileges
* Using GRANT/REVOKE  
  grant privilege-list on relation-name to userlist
* Grant view =/= grant underlying relations
* Grantor must have priv himself
* REVOKE can be on all except those allowed explicitly
* If granted twice, one revoke, still can access
* All privileges dependent on another privilege (will be revoked tgt)

Connection to DB from programming language

* SQL does not provide full expressive power
* Non-declarative actions cannot be done within SQL (print report)

Embedded SQL

* Special pre-processor prior to compilation
* Cannot change on runtime (compiled in advanced)

Dynamic SQL

* Construct + submit SQL queries at runtime
* Needs API

ODBC (Open DB Connectivity)

* ‘driver’ library linked with the client program
* ODBC allocates SQL environment
* Open DB = SQLConnect()
* Prepared Statement (placeholders for values)
* Metadata features (finding relations, etc)
* Treated separately transaction (auto commit)
* Conformance level 1 and 2

JDBC

* Java
* Metadata

Stored Procedures

* Ifelse, for, while
* Execute them using **call** statement

Code details:

* Rset.getString(“branchnname”) & rset.getString(1) same if first argument of result set
* Rs.wasNull()

SQL functions

* Allows overloading (diff arg loaded)
* Allows return of table
* Written as procedure as well

Procedural Constructs

* While and repeat, for loop
* Begin … end

External Language Routines

* More efficient for many ops,

Security concerns

* Code to implement function needed to be loaded in db and exec in DB system
  + Accidental correction
  + Security risk (user can access unauthorized data)
* Alternatives = good security , bad perf

Sandbox or Run external language in separate process (no access to DB memory, talk via IPC)

* Performance overhead

**Relational DB**

* Without unnecessary redundancy, retrieve info
* Functional Dependencies (normal form)
* Spilt tables (decompose)
* Combine tables = repetition of information
* Rule, functional dependency

**Lossy Decomposition**

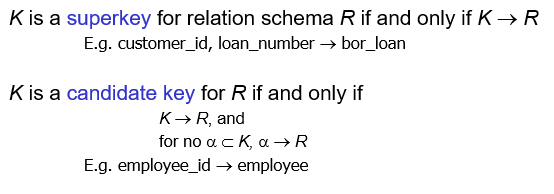
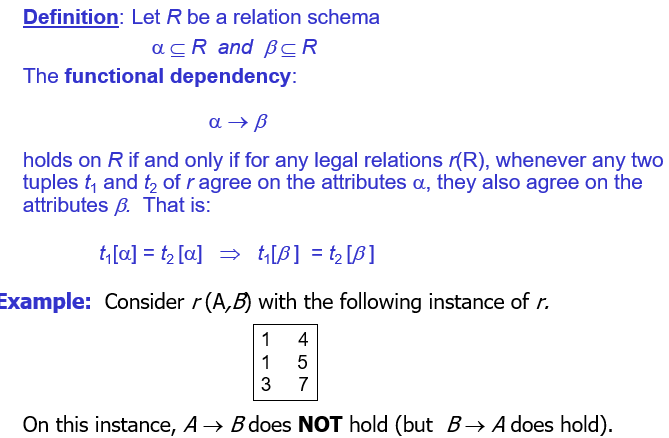
* Lose info, cannot reconstruct original relation

**Good Form**

* Decompose everything into good form
* Lossless-join decomposition
* **Based on:**
  + Functional dependencies
  + Multivalued dependencies

**Functional Dependencies**

* Constraints on set of legal relations
* Notion of key (uniquely identify tuple)



* express constraints on the values of attributes  
  (that cannot be expressed using superkeys)
* allow us to generalize how relations should satisfy certain properties

Loan\_number -> amount = (good)

Amount-> customer\_name = (wrong)

**Used for:**

* Test relations (legal, given set of functional dependencies)
* Specify constraints (on set of legal relations)

\*smt might turn out to be ok, but not legal  
 (amount->cust\_name)

**Closure & Canonical**

Closure = set of ALL functional dependencies logically implied by F is the CLOSURE of F

Closure of F = F+

F+ is a superset of F

**Armstrong’s Axioms**

if *β* ⊆ α, then α → *β* (reflexivity)

if α → *β,* then γ α → γ *β* (augmentation)

if α → *β,* and *β* → γ, then α → γ (transitivity)

**Sound:** generate only functional dependencies that hold

**Complete:** generate all functional dependencies that hold

To compute:

*F* + = *F*  
**repeat**  
 **for each** functional dependency *f* in *F*+  apply reflexivity and augmentation rules on *f* add resulting functional dependencies to *F* + **for each** pair of functional dependencies *f*1and *f*2 in *F* +  **if** *f*1 and *f*2 can be combined using transitivity  
 **then** add the resulting functional dependency to *F* +**until** *F* + does not change any further

**Extraneous Attributes**

Attribute = extraneous if can be removed without changing closure of set of functional dependencies

**Canonical Cover**

Set of FD may have redundant D that can be inferred from others

= minimal set of FD equivalent to F, having no redundant dependencies or redundant parts of dependencies

A ***canonical cover***for *F* is a set of dependencies *Fc* such that

* *F* logically implies all dependencies in *Fc,* and
* *Fc*logically implies all dependencies in *F,* and
* No functional dependency in *Fc* contains an extraneous attribute, and
* Each left side of functional dependency in *Fc* is unique.

Compute:

Fc = F;

**repeat** Use union rule to replace any dependencies in *Fc* α1 → β1 and α1 → β2 with α1 → β1 β2   
 Find a functional dependency α → β in *Fc* with an   
 extraneous attribute either in α or in β   
 If an extraneous attribute is found, delete it from α → β  
**until** *Fc* does not change

**Normalization**

Processing of organizing data to minimize redundancy

TO AVOID

* **Insertion anomalies** (check consistency of info when inserting data)
* **Deletion anomalies** (info may lost as a result of deleting last tuple)
* **Update anomalies** (updates is consistent)

**1st Normal Form**

* Domain is atomic (indivisible)

**Boyce-Codd Normal Form (BCNF)**

* For all functional dependencies
  + α → *β* is trivial (i.e., *β* ⊆ α)
  + α is a superkey for *R*

**BCNF and Dependency Preservation**

All FD holds, decomposition = dependency preservation

No schema that includes all attributes appearing in this functional dependency. = not dependency preserving

**3rd Normal Form**

* For all functional dependencies
  + α → *β* is trivial (i.e., *β* ⊆ α)
  + α is a superkey for *R*
  + Each attribute *A* in *β* – α is contained in a candidate key for *R.*
* *If in BCNF, it is also in 3rdNF*

**Differences of 3NF and BCNF**

3NF

* Decomposition = lossless
* Dependencies preserved

BCNF

* Decomposition = lossless
* May not be possible to preserve dependencies

Goals:

* BCNF
* Decomposition = lossless
* Preferably, should be dependency preserving

If cannot then accept:

* Lack of dependency preserving
* Redundancy due to 3NF

\*Doesn’t work in SQL

Lossless decomposition

= satisfies all the functional dependencies:  


R generated when converting ER diagram to set of tables

R single relation, normalisation due to break into smaller relations

R = result of some adhoc design of relations, then convert to NF

ER carefully designed = no need further normalization

Can denormalize for performance.

**Transactions**

= unit of program execution that accesses and possibly updates various data items

2 main issues = failures, concurrent execution

**Requirements**

1. **Atomicity Req =** partial executed transactions should not be reflected
2. **Durability =** Updates to the DB must persist
3. **Consistency** = includes :  
   **-**specified integrity constraints (PK and FK)

**-**implicit integrity constraints

Preserves consistency of db

1. **Isolation =** executed in a serial manner (even when multiple transaction)

**Transaction state**

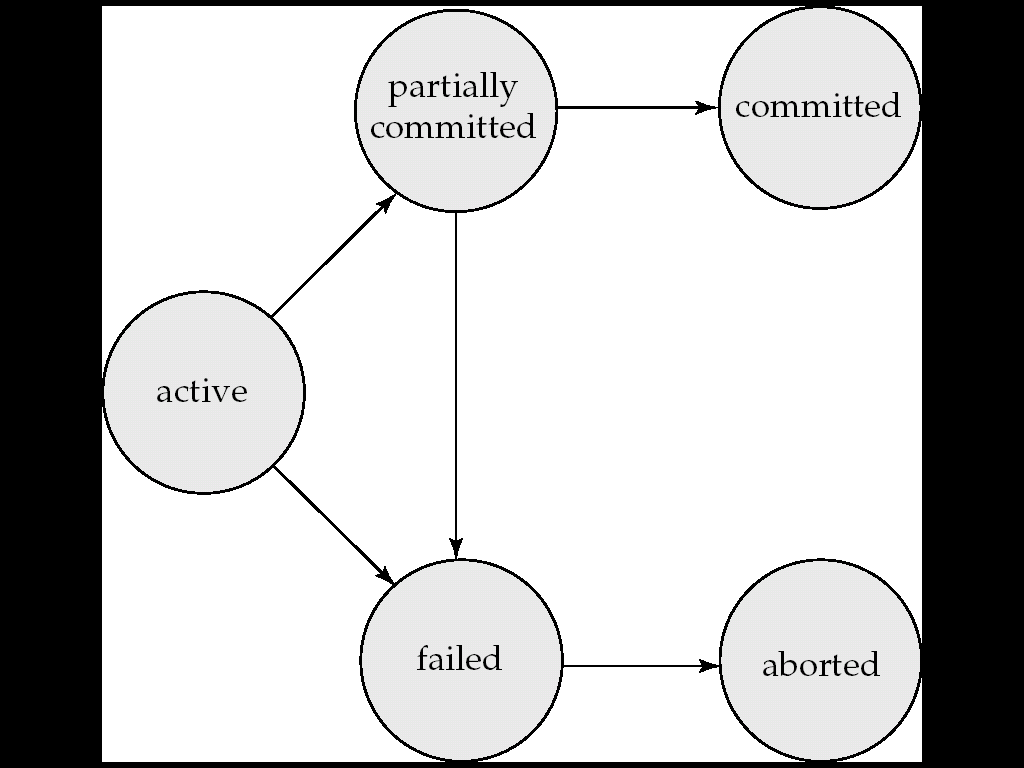
**Active** = while executing

**Partially committed** = after final statement

**Failed** = discovery that normal execution no longer proceed

**Aborted =** rolled back (restart/kill transaction)

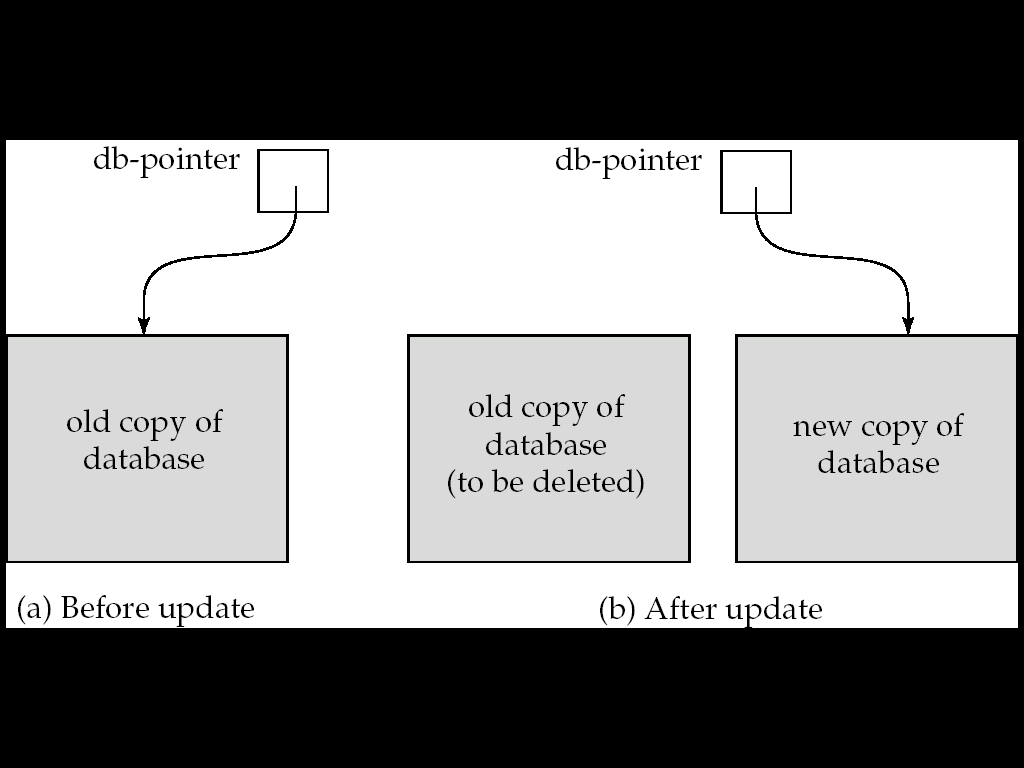
**Committed** = successful completion



**Implementation of Atomicity and Durability**

Recovery-management.

Shadow-database scheme. Updates made on shadow copy, then pointer points to updated shadow copy



**Concurrent Executions**

* Increased processor and Disk utilization
* Reduced average response time

Concurrency control scheme – achieve isolation

Schedule = a sequence of instructions that specifies the chronological order in which instructions of concurrent are executed.

Commit/Abort as last statement

**Serializability**

Some schedules ensure consistency, some don’t

Basic assumption = transaction preserves DB consistency

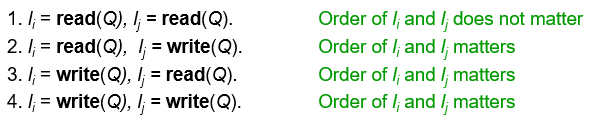
* Serial does it

A possibly concurrent schedule is serializable if it is equivalent to a serial schedule

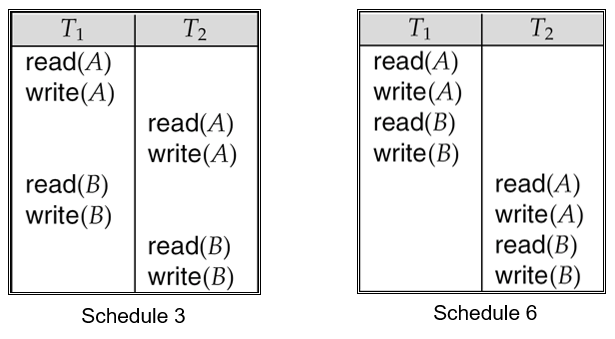
\*ignore operations other than read and write

Other operations happen in buffer

**Conflicting Instructions**

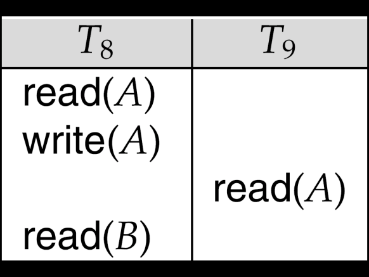


**Conflict equivalent** = swaps of non-conflicting instructions

**Conflict serializable** = swaps -> serial schedule

**Recoverable Schedules**

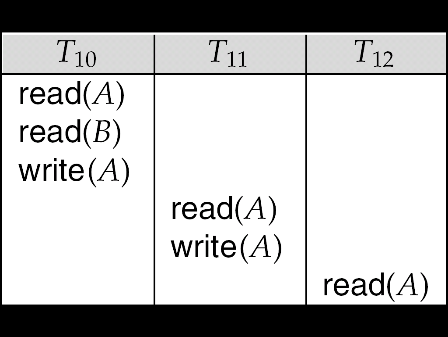
* Need to address effect of trans failure

non – recoverable

* Schedule should be recoverable

**Cascading Rollbacks**

* Single trans failure -> series of trans rollback



* If T10 fails, T11, and T12 must be rollback

Cascadeless Schedules

* Cascading rollback cannot occur
* T10 must commit first, then T11
* Then will be recoverable

Restrict schedules to those cascadeless

For Consistency:

* Serializable
* Recoverable
* cascadeless

**Indexing**

* Access/Insert/Deletion Time
* Space overhead

**Search Key**- attribute to lookup records

Index files = smaller than original

2 kinds of indices

* **Ordered indices**
* **Hash indices**

**Ordered index: Sorted**

**Primary index** = seq. ordered

**Secondary index** = diff order from seq. order of file   
(non-clustering index)

**Dense Index Files**

* Index everything (takes space, not good)

**Spare Index Files**

* Index some search-key values
* Less space
* Less maintenance overhead
* Slower than dense

**Multilevel index**

If primary index doesn’t fit in memory, access = expensive

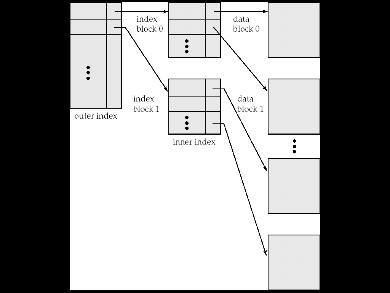
Primary index on disk = seq file, then construct sparse outer index

Outer index = sparse index of primary

Inner index = primary index file

When outer index, too big -> + another level

BUT indices (all levels) -> updated during insert/delete



**Index Update: Deletion**Search key deleted if only record in file with search-key value

Dense – deletion

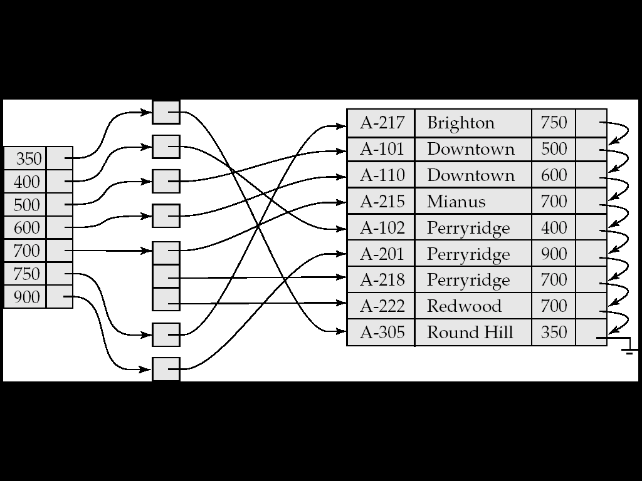
Sparse – if still exists, entry replaced with next value. If not, entry deleted

**Index Update: Insertion**Dense – if don’t have, insert

Sparse – no change is made, unless new block created

**Secondary Indices**common search terms.

must be dense (as it is not ordered)



Indices:

* Substantial benefits
* Overhead on db modification
* Sequential scan on primary = efficient  
  secondary = expensive

B+ Tree Index

Disadvantage of index seq

* Performance drops as file grows
* Periodic reorg required

Adv of BTree

* Automatically reorg itself (when insert/deletion)
* Reorg not required

Minor disadv

* Extra insert and delete overhead, space overhead

Multiple Key Access

* Where there are >2 search terms
* Get 2 search results and then check intersection.
* Less efficient = a lot of results that satisfy 1 cond

Indices on Multiple Keys

* Composite search keys (branch\_name, balance)
* Lexicographic ordering (to sort)

**Hash Function**

**Static Hash**

Bucket = unit of storage containing records

Hash file org = obtain bucket of record from search-key value using hash

Don’t need for index

Different search-key can be same bucket (then searched seq later to locate record)

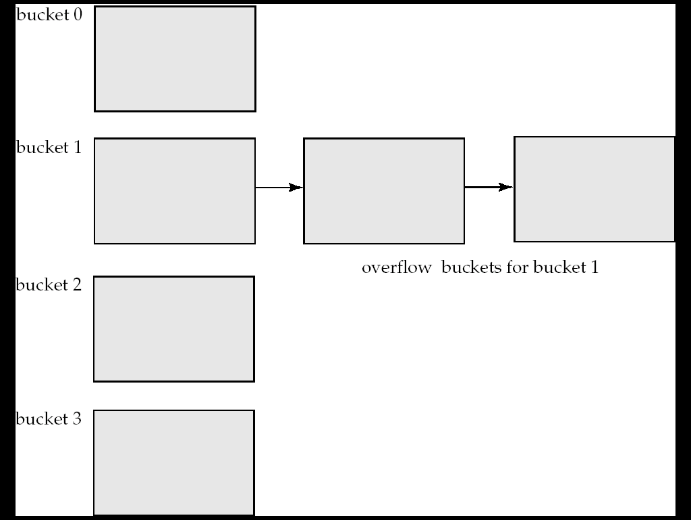
* Worst = all search key -> same bucket
* Ideal = uniform at all buckets, random

**Bucket Overflow**

* Insufficient bucket
* If static hash not ideal
* Probability can be reduced, not eliminated

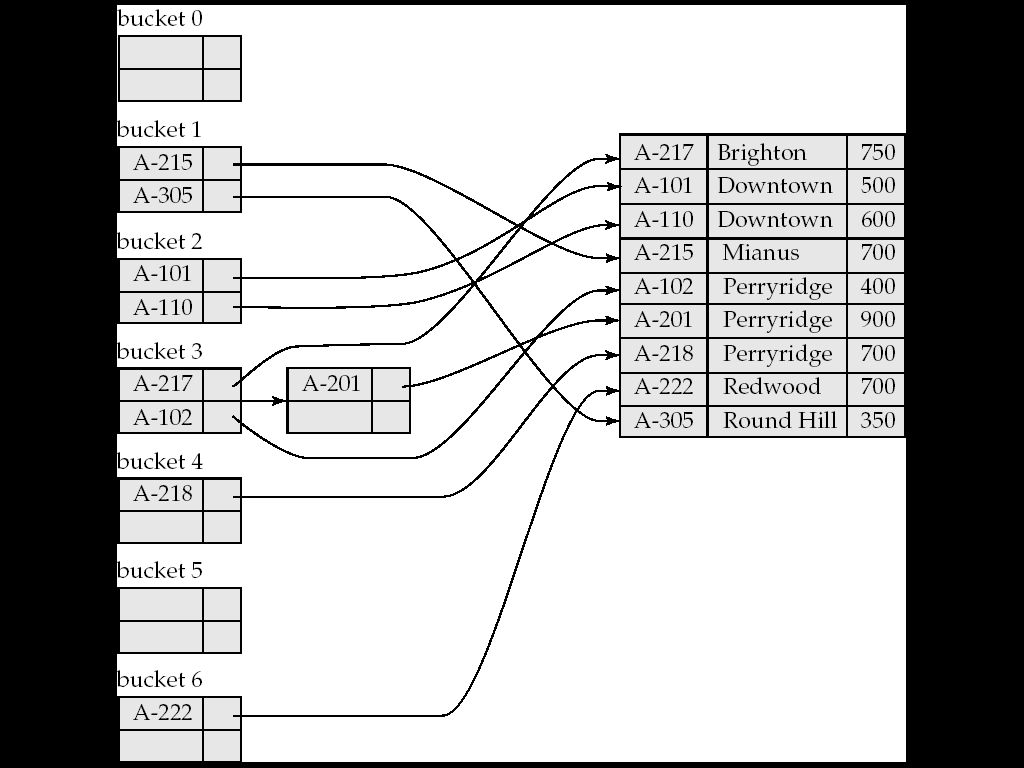
Bucket chaining

* Closed hashing



**Hash index** = organize search keys, with associated record pointers.

* Like primary hash, secondary indices



**Dynamic hashing**

Why?

* DB grows and shrink
* No. of buckets = small, file grows, perf drops
* No. of buckets = high, amount of space wasted

Can keep reorganizing, but expensive

Solution = Extendable hashing

Hash = long, 2i = 0-32 , increase i accordingly when grows

Index VS Hashing

* Cost of periodic reorganization
* Freq of insertion/deletion
* Optimize average access time VS worst possible access time

Specified value of key = Hashing better

Range queries is common = ordered indices better

SQL: Create index <index-name> on <relation-name> (<attribute-list>)

**Dynamic Hash Algo**

**Aim:**

* Cope with changes in DB size by splitting and coalescing (grow and shrink)
* Space efficiency retained
* Resulting perf overhead = low

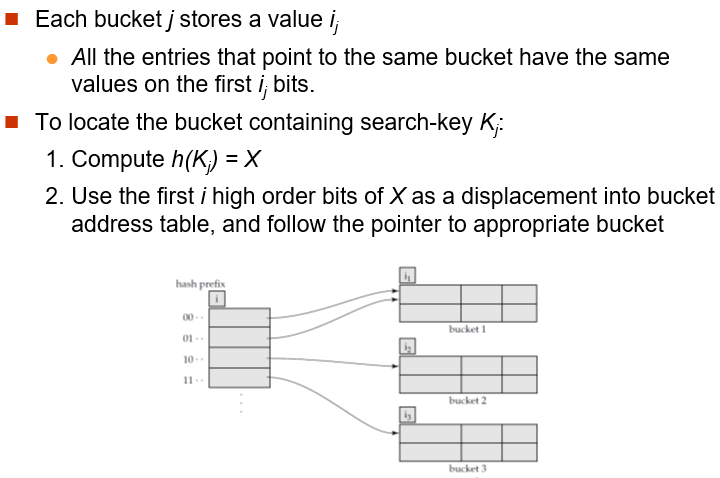
**Data Struct:**

* Use hash function (h)
* (h) -> 32 bit string (can be larger)

Have bucket address (so hash doesn’t lead directly to bucket)

Create buckets on demand, as records are inserted (not bucket for each value)

If i = 2, then we have table of 2^2 size, 4 possible prefix

Lookup Operation:

Insert:

* same as lookup, find the bucket
* room in bucket, then insert
* else bucket split, insert reattempt
  + maybe use overflow bucket   
    (if increase (i) value wont change anything)