Revision Lecture on Selected Topics

Thomas Heinis

t.heinis@imperial.ac.uk
Scale Lab - scale.doc.ic.ac.uk

Imperial College London

ER Modeling

Entities, Relationships, Attributes

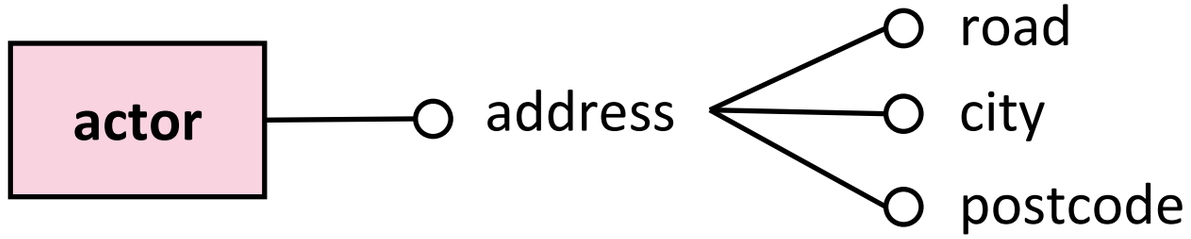
The three main elements in an ERD are entity sets, relationship sets and attributes

Entity sets	An entity set is a set of distinguishable entities that share the same set of properties. Entities could be *physical* (e.g. car, room), an *event* (e.g. car sale, flight), or *conceptual* (e.g. goodwill). Normally correspond to **nouns**.	Rectangles
Relationship	A relationship set captures how **two or more** entity sets are related to one another (e.g. owns, tutors). Sometimes correspond to **verbs**. We can have more than one relationship set between entity sets (e.g. owns and drives). We can also have a relationship set on the same entity set (e.g. supervises). A relationship is a particular instance.	Diamonds
Attributes	Attributes are the properties of an entity (e.g. name, price). **Primary key attributes are underlined**. Relationship Sets can also have attributes (e.g. owned since). This is avoids putting them in an entity set.	Small Circles

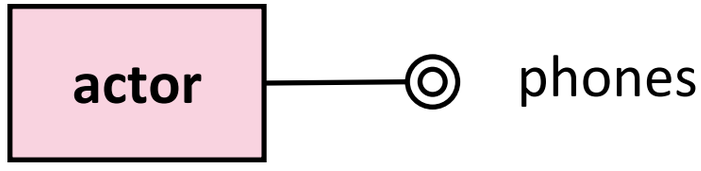
Complex Attributes

In E-R modelling, attributes need not be simple. Simple attributes are atomic, e.g. year, first name, credit card number. Complex attributes can be:

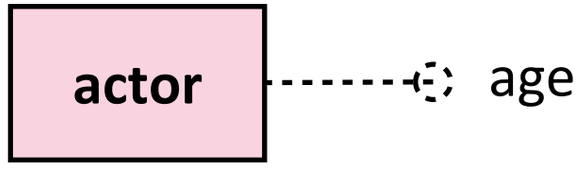
Composite. These can be subdivided, e.g. address into road, city, postcode



Multivalued. A set of values, e.g. several phones, supervisors



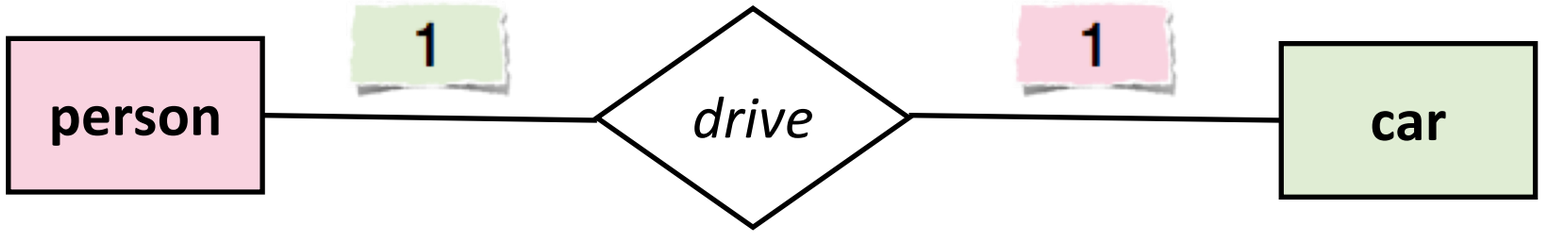
Derived. Computed from other values, e.g. age computed from date of birth



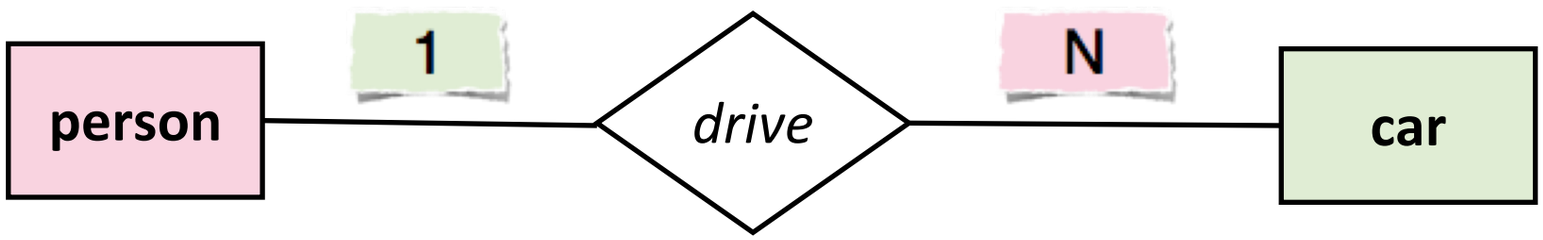
Cardinality Constraints

In E-R modelling, it is important to impose constraints on the number of entities that can be in a relationship. A relationship between two entity sets (a **binary relationship**) can be:

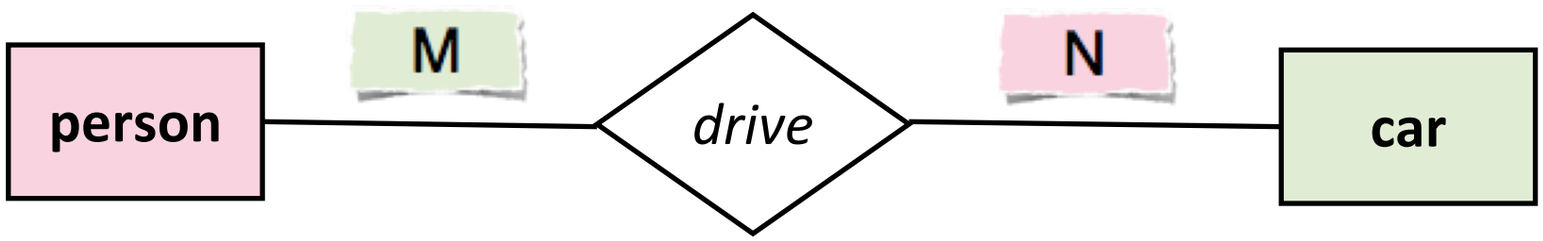
One-to-One. A person can drive 1 car. A car can be driven by 1 person.



One-to-Many. A person can drive **many** (N) cars. A car can be driven by 1 person.

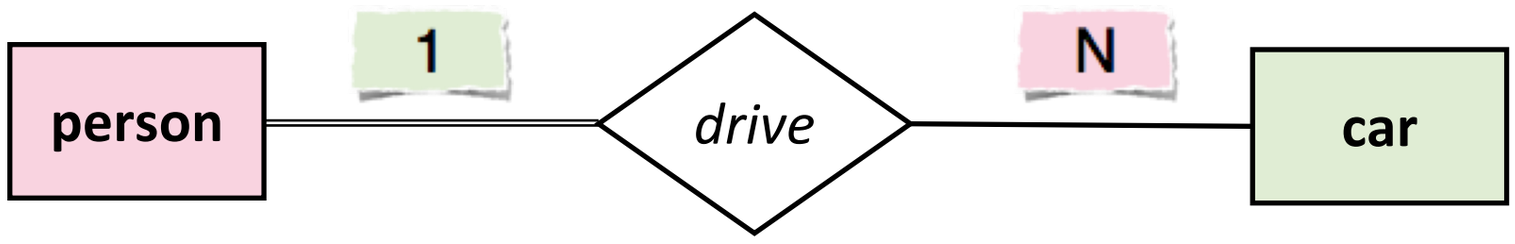


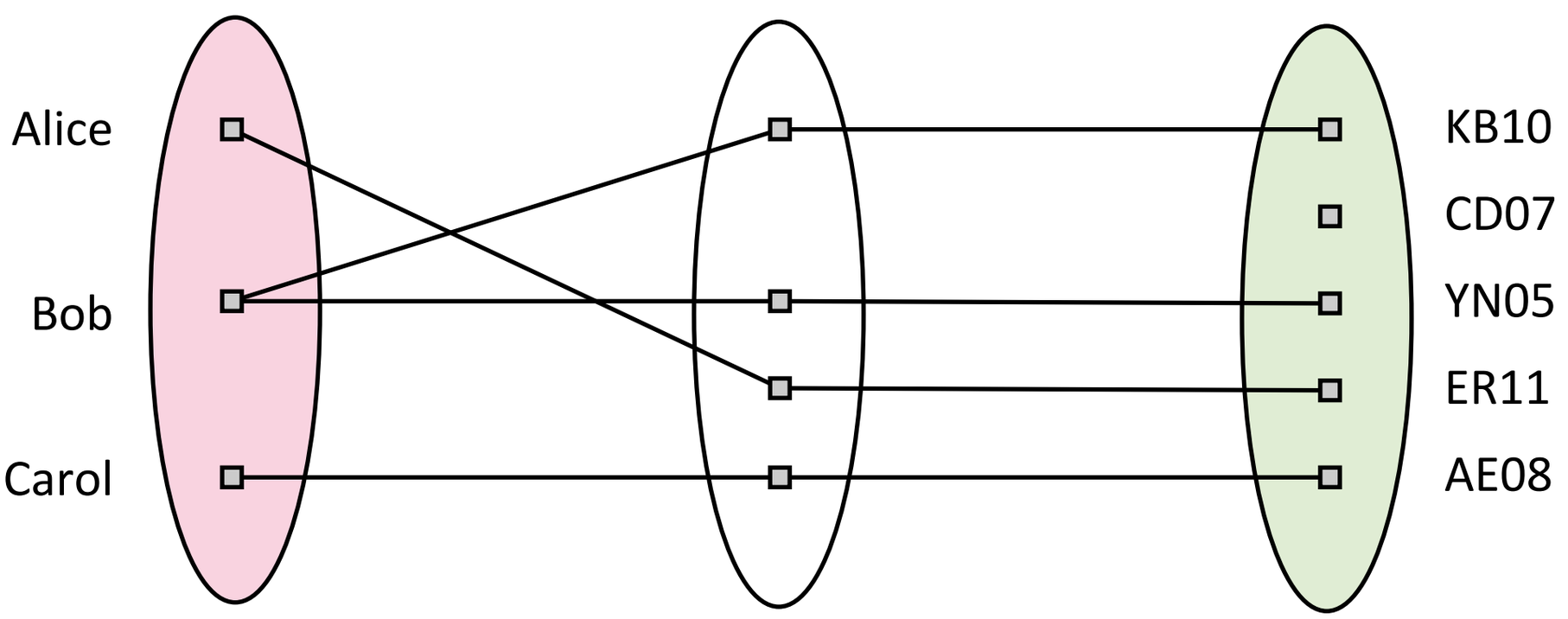
Many-to-Many. A person can drive **many** cars. A car can be driven by **many** people.



Entity Set Participation

We can indicate that all entities in an entity set must participate in a relationship by using a double line from the entity set to the relationship set. This is called **total participation**.



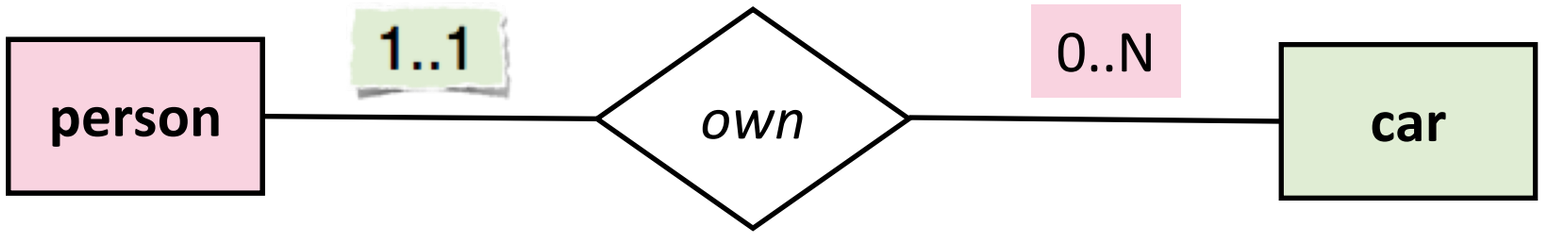


Every person drives at least one car. A car can have at most 1 driver.

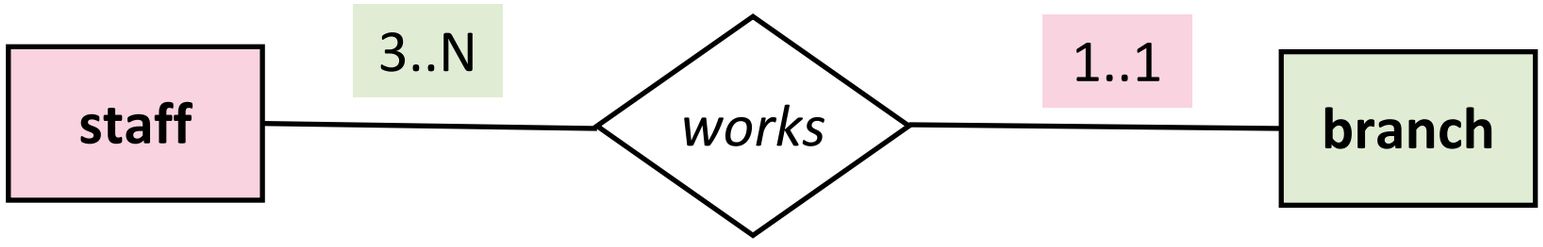
Participation Bounds

Rather than double lines some E-R notations allow explicit bounds on the degree of participation:

People can own 0 or more cars. Every car must have 1 owner.



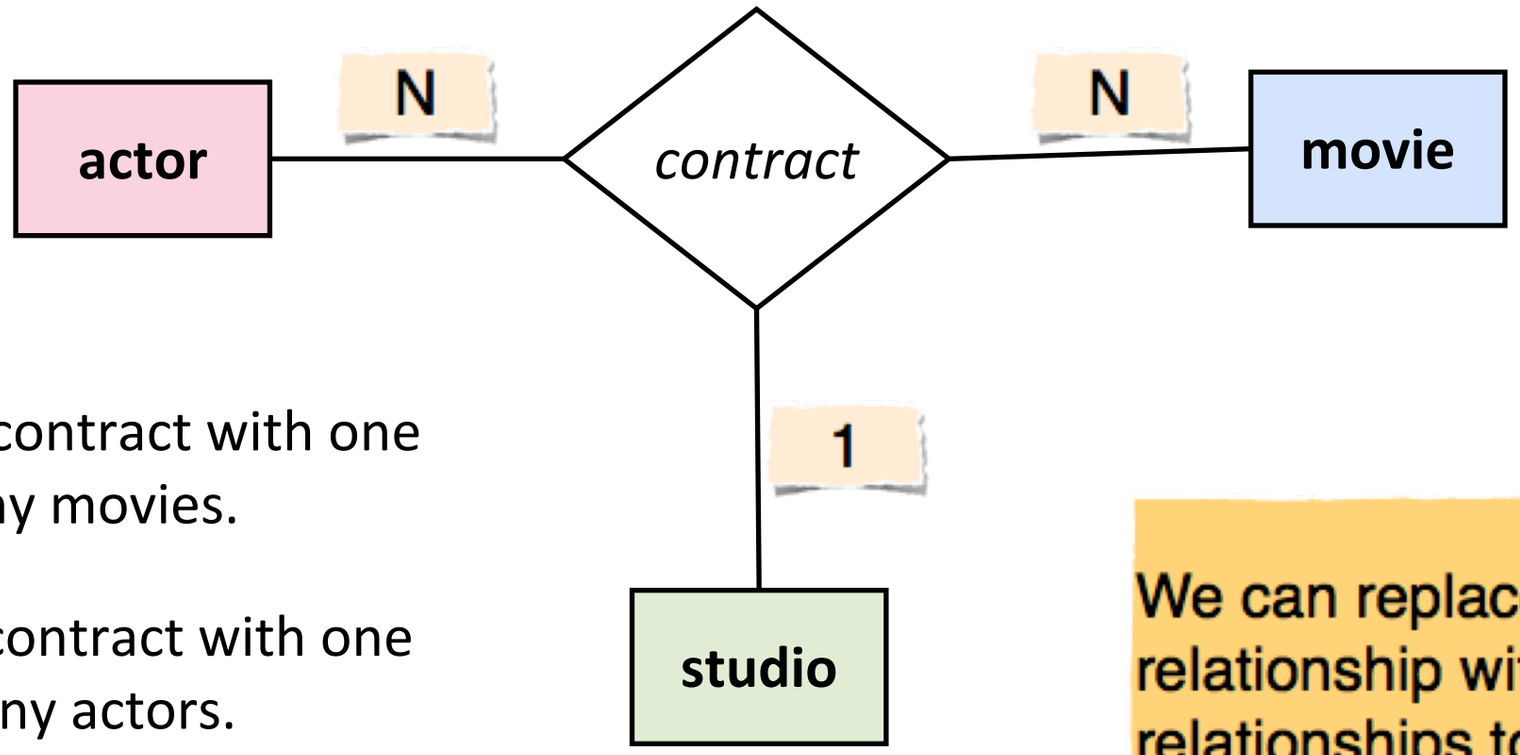
Staff work in 1 branch. Each branch must have at least 3 members of staff.



Note: In some ER notations, cardinality bounds are shown next to the entity set (**look here**) rather than on the other side of the relation (**look there**).

Multiway Relationships

Although rarer, a relationship can involve more than two entity sets:



An actor may contract with one studio for many movies.

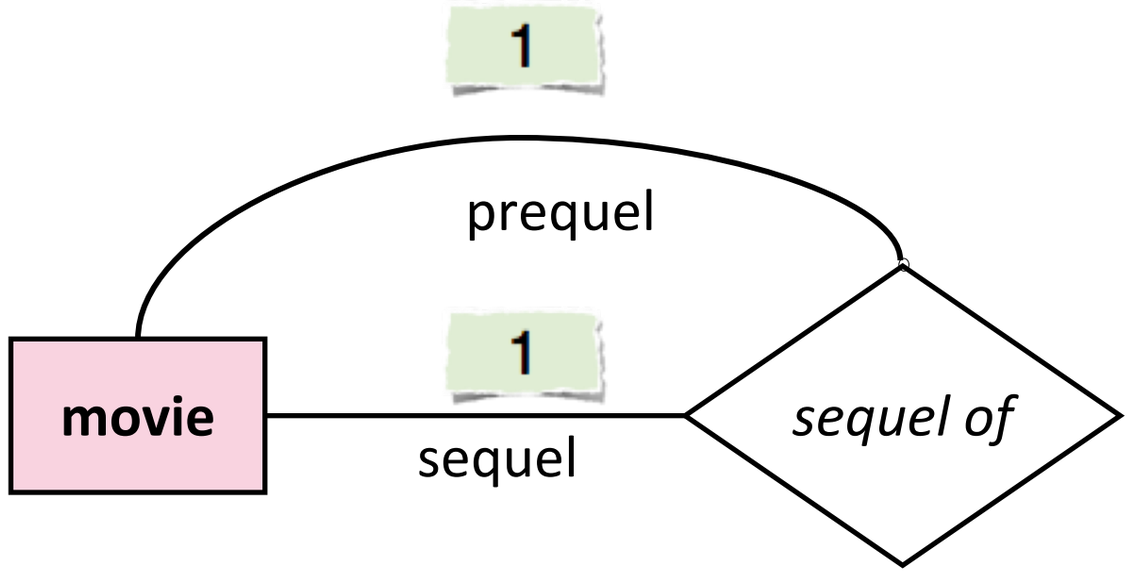
A movie may contract with one studio and many actors.

A studio may contract with many actors for many movies.

We can replace a ternary relationship with 3 binary relationships to a new entity set for the ternary relationship. We can't always translate cardinality constraints, however.

Roles in Relationships

If an entity set plays more than one role in a relationship, then we draw several lines from the entity set to the relationship set and label each line with the role.

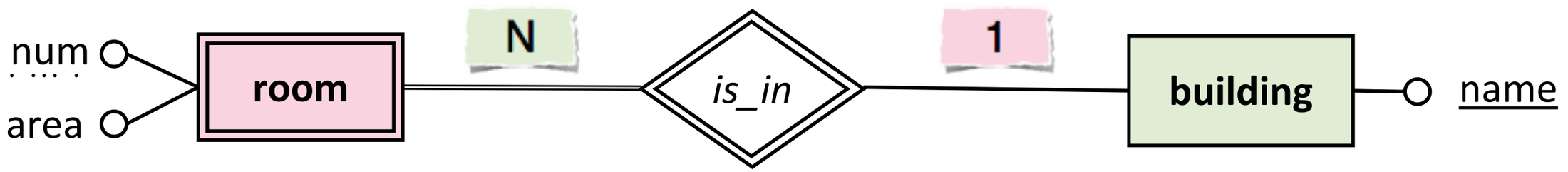


The *sequel of* relationship is between two movies, one of which is a sequel of the other.

A movie can have one prequel and one direct sequel.

Weak Entities

Entities which cannot be uniquely identified using their own attributes are called **weak entities**, in contrast to entities that have a primary key which are known as **strong entities**. A weak entity is dependent on a strong entity for its existence. If a strong entity is deleted, any dependent weak entity would also have to be deleted.

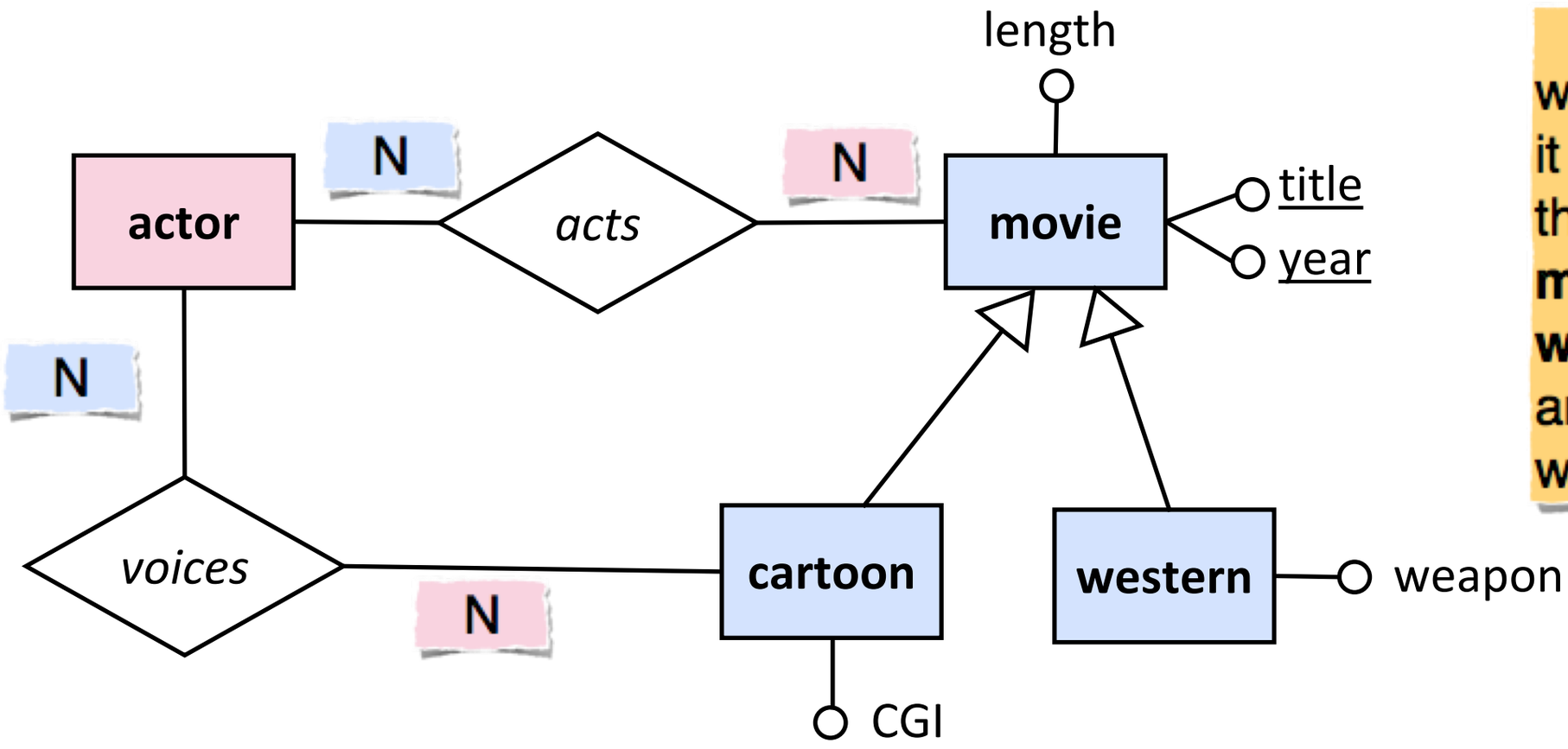


We indicate weak entity sets using a *double rectangle*, and the associated relationship to the strong entity set using a *double diamond*. The primary key for a weak entity set is formed using the primary key of the strong entity and one or more attributes of the weak entity set (shown with a *dashed underline*).

Note: The weak-strong entity relationship is always many-to-one from the weak-entity set to the strong entity set with the total participation of the weak entity set.

is-a Relationships

Specialisation/generalisation is represented in an ERD using *is-a* relationships. We'll represent *is-a* using lines with a hollow arrow-head:



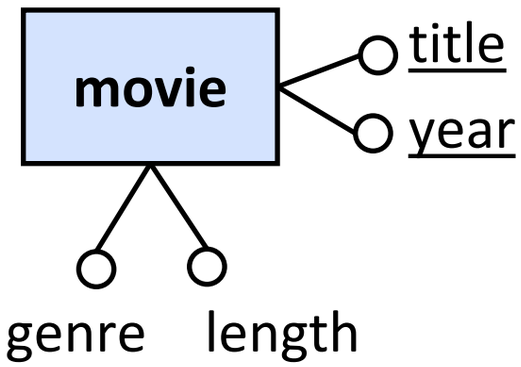
western *is-a* **movie**, it has (*inherits*) all the attributes of **movie** plus **weapon**, it also has an *acts* relationship with **actor**.

cartoon *is-a* **movie**, it has all the attributes of movie. It also has an *acts* relationship with **actors** and a *voices* relationship with **actor**.

Translation ER to RM

Strong Entity Sets with Simple Attributes

A strong entity set with simple attributes can be mapped directly to a relation with the same attributes. Each tuple in the relation would correspond to an entity in the entity set. The primary key of the entity set becomes the primary key of the relation.

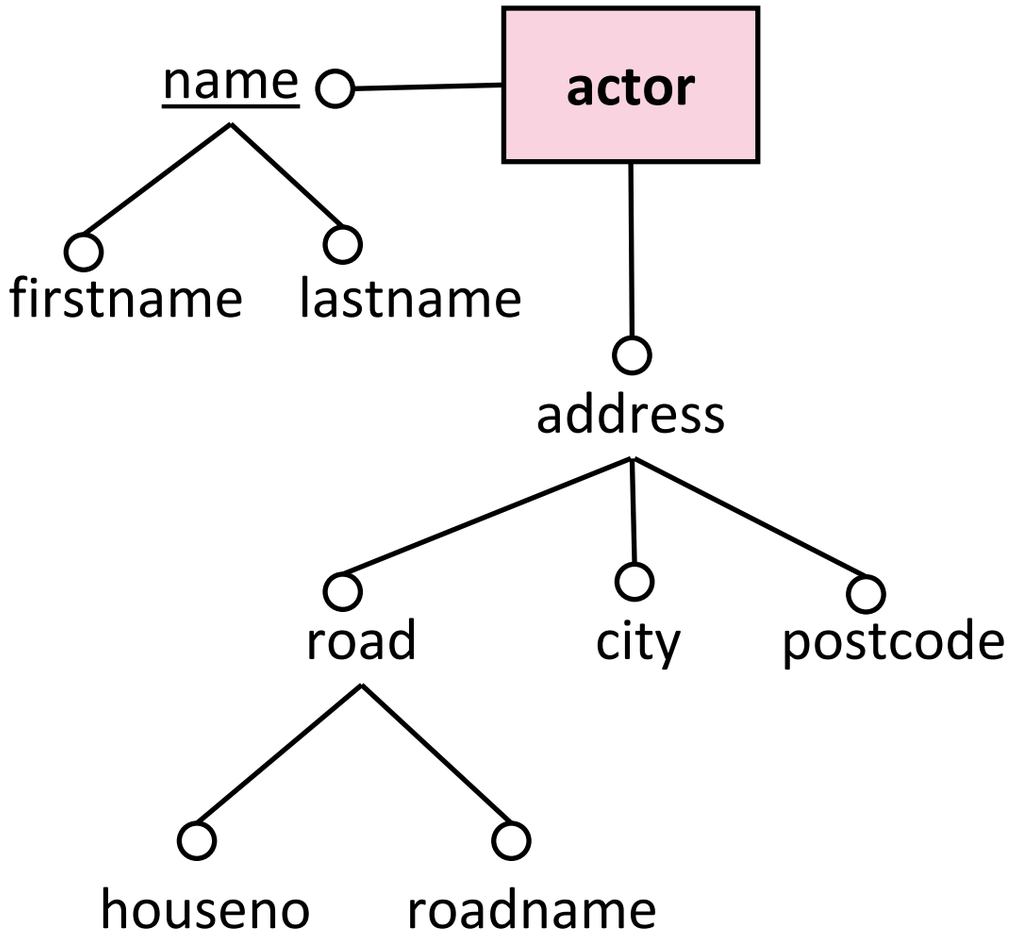


movie(<u>title</u>, <u>year</u>, length, genre)

```
create table movie (
        title       varchar(120),
        year        int,
        length      int,
        genre       char(20),

        primary key (title, year)
)
```

Composite Attributes

If an entity set has a composite attribute, we ‘flatten’ the attribute. That is, the mapped relation includes only the simple attributes within the composite attribute.

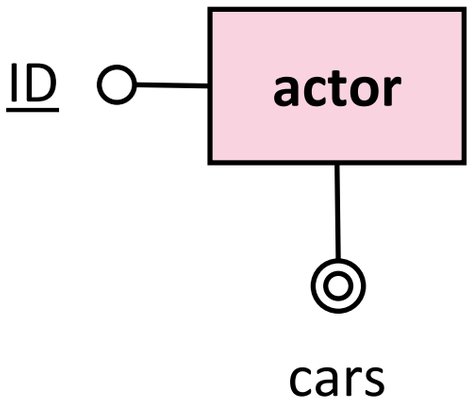


actor(<u>firstname</u>, <u>lastname</u>, houseno, roadname, city, postcode)

```
create table actor (
        firstname       varchar(30),
        lastname        varchar(30),
        houseno         int,
        roadname        varchar(30),
        city            varchar(40),
        postcode        varchar(10)

        primary key (firstname, lastname)

)
```

Multivalued Attributes

Multivalued attributes are mapped to their own relation and linked back to the entity set relation using a foreign key constraint:



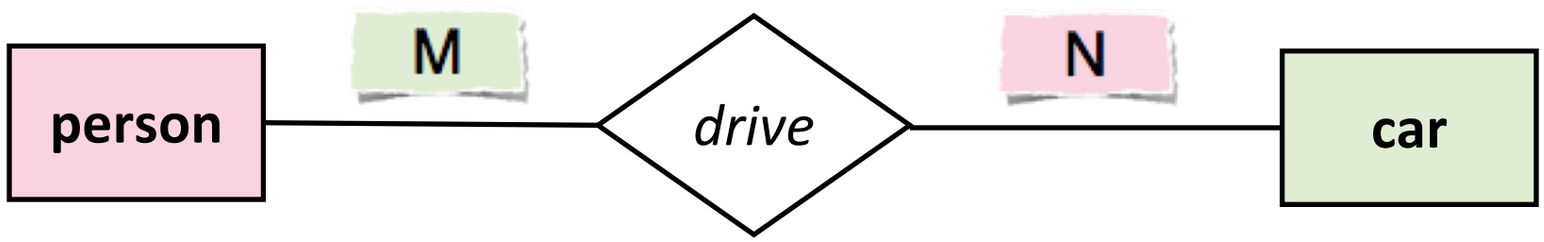
actor(<u>ID</u>, *otherattributes*)

actor_cars(<u>actorID</u>, <u>carID</u>, *otherattributes*)

```
create table actor_cars (
        actorID    int,
        carID      varchar(10),

        primary key (actorID, carID),
        foreign key (actorID) references actor.ID
)
```

Many-to-Many Relationship Sets

We map a Many-to-Many (binary) Relationship set into a relation with two foreign keys, corresponding to the primary keys of each entity set involved in the relationship. The primary keys of the entity sets form the primary key of the relationship set relation.



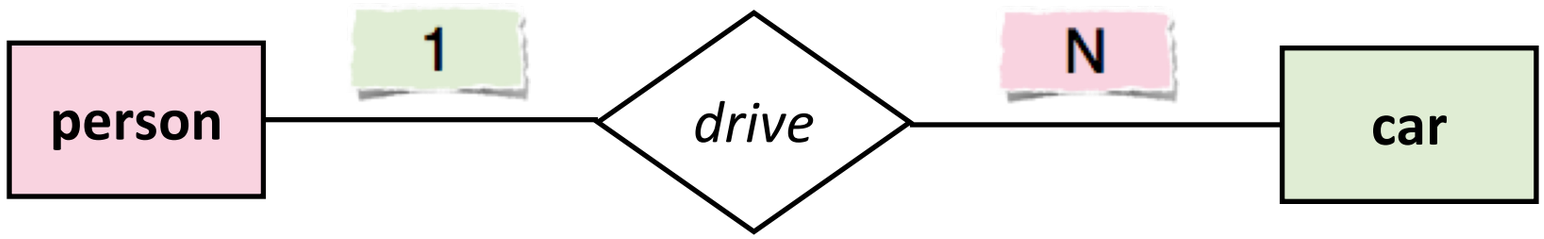
person(<u>ID</u>, *otherattributes*)
car(<u>regno</u>, *otherattributes*)
drive(<u>personID</u>, <u>regno</u>,*otherattributes*)

If the relationship set has its own attributes, they're included here, e.g isPolicyholder

```
create table drive (
        personID  varchar(10),
        regno       varchar(12),
        primary key (personID, regno),
        foreign key (personID) references person.ID on delete cascade,
        foreign key (regno) references car.regno on delete cascade

)
```

One-to-Many Relationship Sets

A One-to-Many Relationship set can be mapped like a Many-to-Many Relationship. However, it's simpler to directly include the primary key of the One relation as a foreign key attribute in the Many relation:



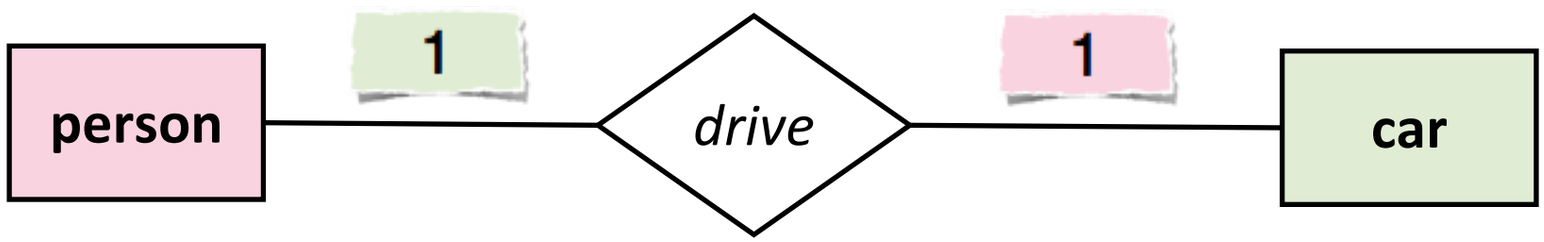
person(<u>ID</u>, *otherattributes*)
car(<u>regno</u>, personID, *otherattributes*)

```
create table car (
        regno      varchar(12),
        personID varchar(10),
        primary key (regno),
        foreign key (personID) references person.ID
)
```

If the relationship set has its own attributes, they're included in the Many relation.

One-to-One Relationship Sets

Like a One-to-Many Relationship we directly include the primary key of one of the One relations as a foreign key attribute in the other One relation. The choice is left to the database designer:



person(<u>ID</u>, <u>regno</u>, *otherattributes*)
car(<u>regno</u>, *otherattributes*)

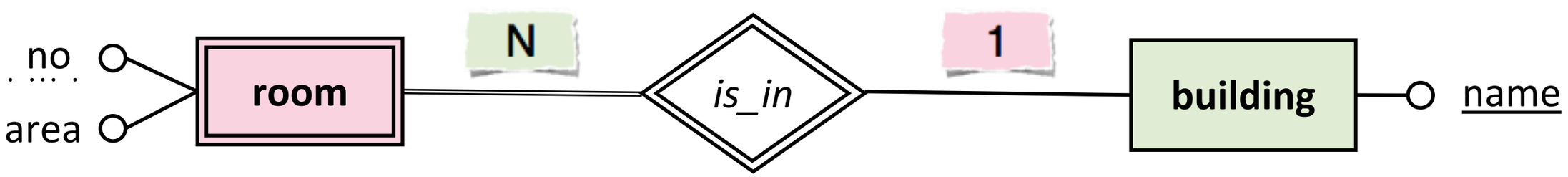
```
create table person (
        personID  varchar(12),
        regno     varchar(10),
        primary key (personID),
        foreign key (regno) references car.regno
)
```

Note: We can use the mapping on the previous slide also

If the relationship set has its own attributes, they're included in the relation with the foreign key.

Weak Entity Sets

Like strong entity sets, a weak entity set is mapped to its own relation and attributes but includes the primary key of the strong entity set as a foreign key with an on delete cascade constraint. The relationship isn't mapped.



building(<u>name</u>, *otherattributes*)
room(<u>no</u>, <u>buildingname</u>, *otherattributes*)

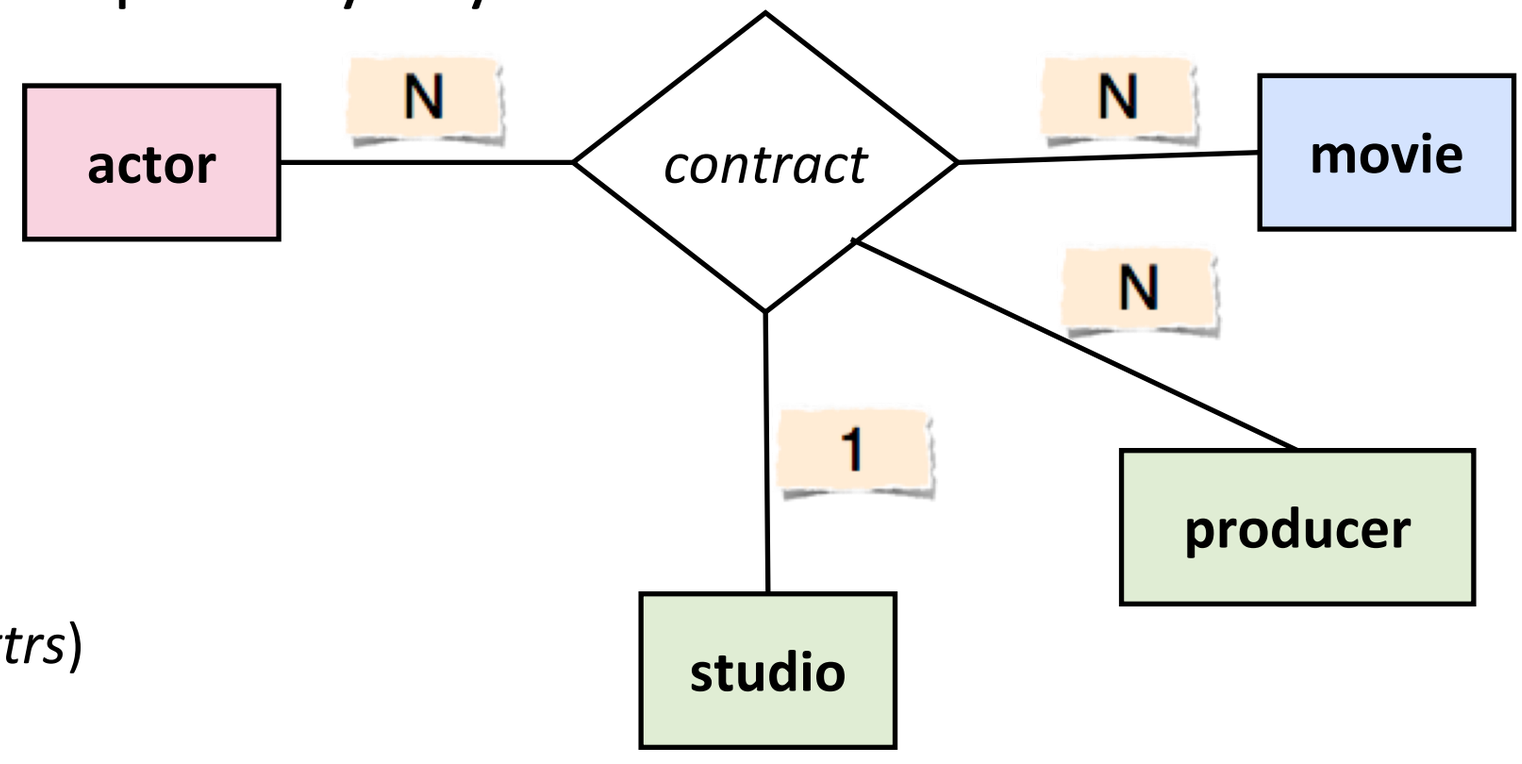
```
create table room (
        no    varchar(120),
        buildingname varchar(50) not null,

        primary key (no, buildingname),
        foreign key (buildingname) references building.name on delete cascade
)
```

Ensures total participation.

Note: The primary key for the Weak Entity set relation consists of the primary key of the strong entity set relation and one or more discriminating attributes from the weak entity set

Multiway Relationship Sets

We can map an n-way relationship set into several binary relationships or generalise the many-to-many relationship mapping to include primary keys from all the entity sets as foreign keys. The foreign keys of the Many entity sets form the primary key.



actor(ID, *otherattributes*)
movie(ID, *otherattributes*)
studio(ID, *otherattributes*)
producer(ID, *otherattributes*)
contract(aID, mID, sID, pID,*otherattrs*)

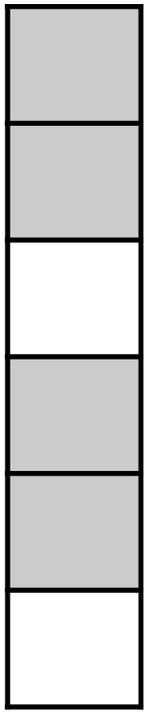
create table contract (
    actorID **int**, movieID **int**, studioID **int**, producerID **int**,
    primary key (actorID, movieID, producerID),
    plus foreign key declarations for actorID, movieID, studioID, producerID
)

Relational Algebra

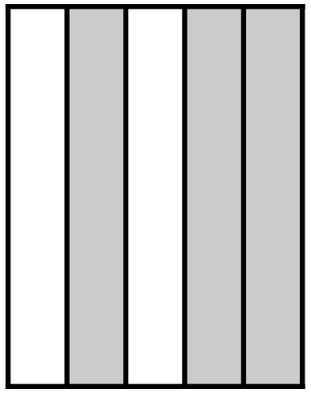
Relational Algebra

Relational expressions are used to construct new relations from other relations. Operators *include*:

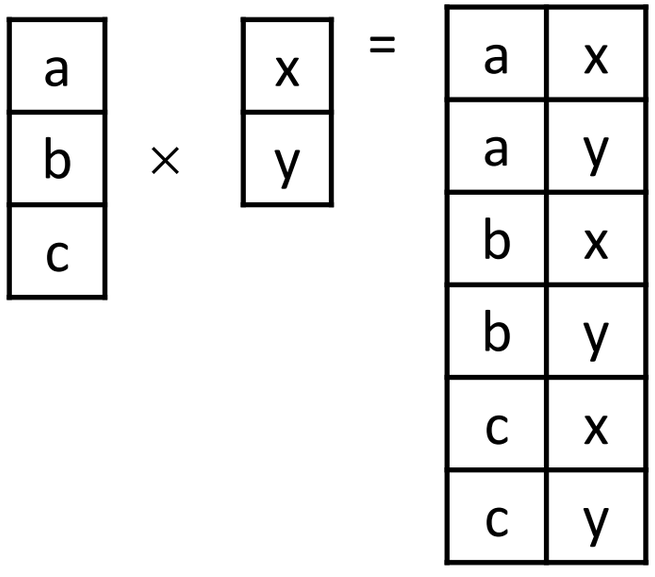
selection



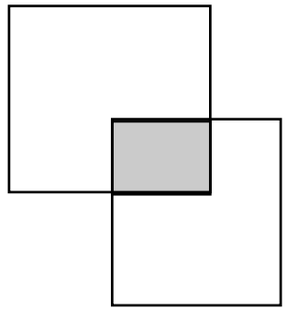
projection



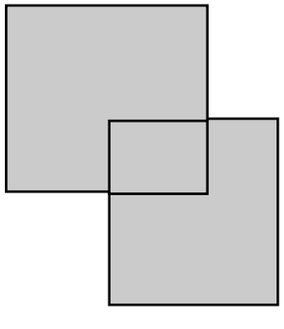
product



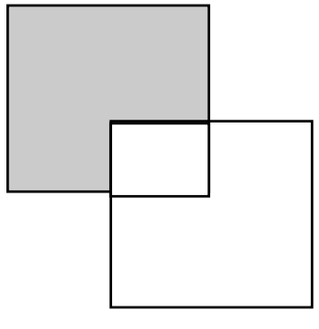
intersection



union



difference



Solution: Cartesian Product R × S

If a R and S have attribute names in common, then we distinguish them by prefixing the attribute name with the name of the relation.

R

a	b
1	2
3	4

S

b	c	d
1	2	3
4	5	6
7	8	9

R × S

a	R.b	S.b	c	d
1	2	1	2	3
1	2	4	5	6
1	2	7	8	9
3	4	1	2	3
3	4	4	5	6
3	4	7	8	9

Natural Join R ⋈ S

Resulting schema is the set of both schemas.

R

a	b
1	2
1	3
3	4

S

b	c	d
1	2	3
2	5	6
4	8	9
4	9	9

R ⋈ S

a	b	c	d
1	2	5	6
3	4	8	9
3	4	9	9

Resulting relation has all tuples that can be "joined" using all matching attributes of R and S.

Left Outer Join R ⟕ $_{R.c=S.b}$ S

R

a	b	c
1	2	2
6	7	8
9	2	2
4	2	7
1	3	8

S

b	c	d
2	2	3
2	3	6
4	8	9
7	8	3

R ⟕ $_{R.c=S.b}$ S

a	b	R.c	S.b	c	d
1	2	2	2	2	3
1	2	2	2	3	6
9	2	2	2	2	3
9	2	2	2	3	6
4	2	7	7	8	3
6	7	8			
1	3	8			

Right Outer Join R ⟖ R.c=S.b S

R

a	b	c
1	2	2
6	7	8
9	2	2
4	2	7
1	3	8

S

b	c	d
2	2	3
2	3	6
4	8	9
7	8	3

R ⟖R.c=S.b S

a	b	R.c	S.b	c	d
1	2	2	2	2	3
1	2	2	2	3	6
9	2	2	2	2	3
9	2	2	2	3	6
4	2	7	7	8	3
			4	8	9

Full Outer Join R ⟗ $_{R.c=S.b}$ S

R

a	b	c
1	2	2
6	7	8
9	2	2
4	2	7
1	3	8

S

b	c	d
2	2	3
2	3	6
4	8	9
7	8	3

R ⟗$_{R.c=S.b}$ S

a	b	R.c	S.b	c	d
1	2	2	2	2	3
1	2	2	2	3	6
9	2	2	2	2	3
9	2	2	2	3	6
4	2	7	7	8	3
6	7	8			
1	3	8			
			4	8	9

Functional Dependencies

Functional Dependency

A Functional Dependency (FD) is a constraint that if two tuples of a relation R agree on a set of attributes A_1, A_2 ... An then they must also agree on the set of attributes B_1,B_2 ... Bm. We write this as:

$$A_1\ A_2 \ldots An \rightarrow B_1\ B_2 \ldots B_m$$

We say that B_1 .. B_n are ***functionally dependent*** on A_1 ... A_n, or
A_1 .. A_n ***functionally determine*** B_1 ... B_m

i.e., for any set of values for A_1 to A_n there is only one set of values for B_1 to B_m.

Functional Dependencies allow the database designer to express constraints over relations and to identify situations were relations ought to be decomposed into two or more schemas.

There are several normal forms that a database designer can use and are defined in terms of functional dependencies, e.g. BCNF, 3NF.

Keys: Superkey, Candidate Key, Primary Key

If a set of attributes {A1,A2 ... An} functionally determines all the other attributes of the relation, we call the set of attributes a **superkey**. This implies that we cannot have two tuples that have the same superkey values (*Uniqueness property*). That is, we're asserting the functional dependency:

superkey → B1 B2 ... Bn where B1 B2 ... Bn are all the other attributes

Note: A relation could have several superkeys. Also a superkey could contain extraneous attributes that are not strictly needed.

We are mostly interested in superkeys for which there is no proper subset of the superkey (*Irreducibility property*). Such a minimal superkey is called a **candidate key** (or just a **key**).

If there is more than one candidate key, then we can choose one to act as the **primary key**. This is important in a RDBMS but not in functional dependency theory.

Splitting and Combining FDs

Splitting Rule FD → FD set. We can replace a FD with a set of FDs with one FD for each attribute of the RHS (Right Hand Side) keeping the LHS (Left Hand Side)

A B C D E → W X Y	A B C D E → W A B C D E → X A B C D E → Y

Combining Rule FD set → FD. Similarly with can replace a set of FDs with the same LHS with a single FD that combines the attributes of the RHSs of the FD set.

A B C D E → W A B C D E → X A B C D E → Y	A B C D E → W X Y

Trivial Dependency Rule

Trivial Dependency Rule. If some attributes on the RHS of a FD are also on the LHS of the FD then we can simplify the FD by removing them from the RHS.

A B C D E → A C X Y Z	A B C D E → X Y Z

A **Trivial FD** is a FD were all the attributes on the RHS are also on the LHS, i.e.
RHS ⊆ LHS

For example, A B C D E F G H I J → A C E J

We can assume any trivial FD regardless of other FDs.

Closure of Attribute Sets

We call the set of all attributes functionally determined by a set of attributes L, under a set of functional dependencies F, the **closure** of L under F, and denote it L^+.

Checking if L is a superkey of relation R

If L^+ contains all the attributes of R then L is a superkey of R.

Checking if a FD LHS → RHS holds

If RHS $\subseteq$ LHS^+ (without LHS → RHS) then LHS → RHS holds

Armstrong's Axioms

Provides us with a **sound** (do not generate any incorrect FDs) and **complete** (allow us to derive all valid FDs) axiomatisation of FDs.

A, B, C .. are attributes. α, β, γ are sets of attributes.

Reflexivity (Trivial FDs)	$\alpha \rightarrow \beta$ always holds if $\beta \subseteq \alpha$
Augmentation	If $\alpha \rightarrow \beta$ then $\alpha\,\gamma \rightarrow \beta\,\gamma$
Transitivity	If $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$ then $\alpha \rightarrow \gamma$

Additional Rules

The following additional rules can simplify the reasoning. They can be derived from Armstrong's Axioms.

Union	If $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$ then $\alpha \rightarrow \beta\,\gamma$
Decomposition	If $\alpha \rightarrow \beta\,\gamma$ then $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$
Pseudotransitivity	If $\alpha \rightarrow \beta$ and $\delta\,\beta \rightarrow \gamma$ then $\delta\,\alpha \rightarrow \gamma$

Closure of a FD set

As well as the closure of attributes, we can also work out the **closure of a FD set** F, i.e. the set of all FDs that are can be inferred, we denote this closure $\mathbf{F^+}$

For example for $A \rightarrow B$, $B \rightarrow C$ we can infer $A \rightarrow C$

One approach for achieving this is to repeatedly apply Armstrong's axioms:

1. Initialise F^+ to the FD set F

2. Apply the reflexivity and augmentation axioms. Add new FDs to F^+

3. Apply the transitivity axiom to suitable FDs in F^+ adding new FD to F^+

4. Repeat from step 2 until no further changes to F^+

Covers of FD Sets

FD Sets F1 and F2 are **equivalent** if each implies the other, i.e., any relation instance satisfying F1 also satisfies F2. F1 and F2 are said to be **covers** of each other.

A cover is said to be **canonical** (or **minimal** or **irreducible**) if

1. Each LHS is unique.

2. We cannot delete any FD from the cover and still have an equivalent FD set.

3. We cannot delete any attribute from any FD and still have an equivalent FD set.

Effectively the canonical cover has no redundant FDs or FD attributes.

Testing if a FD Attribute is Extraneous

LHS Attribute X

LHS X is extraneous if RHS $\subseteq$ {LHS-X}$^+$ under the FD .

B extraneous in A B $\rightarrow$ C under { A B $\rightarrow$ C, A $\rightarrow$ C}?, Yes because C $\subseteq$ { A }$^+$ = {AC}
A extraneous in A B $\rightarrow$ C under { A B $\rightarrow$ C, A $\rightarrow$ C}?, No because C $\not\subseteq$ { B }$^+$ = {B}

RHS Attribute X

RHS X is extraneous if X $\in$ LHS$^+$ under the *FD set with X removed from the RHS*.

C extraneous in A B $\rightarrow$ C D under { A B $\rightarrow$ C D, A $\rightarrow$ E, E $\rightarrow$ C},? Yes, **C** $\in$ { AB }$^+$={ABDEC}

D extraneous in A B $\rightarrow$ C D under { A B $\rightarrow$ C D, A $\rightarrow$ E, E $\rightarrow$ C}?, No, **D** $\notin$ { AB }$^+$={ABCE}

Computing a Canonical Cover

To compute a canonical cover F for an FD set we can use the following algorithm:

F := FD set

repeat over F

Replace dependencies of the form $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$ with $\alpha \rightarrow \beta\gamma$ (UNION RULE)

Remove all extraneous attributes one at a time.

until F doesn't change.

Normalisation

Decomposition Properties

When decomposing a relation it is also important to try ensure that we can recover the original relation by *joining* the decomposed relations **and** also preserving the FDs of the original relation.

Lossless Decomposition

If a relation R is decomposed into 2 relations S and T, the decomposition is **lossless** if at least one of the following FDs holds in the closure of the FD set of R:

$$attr(S) \cap attr(T) \rightarrow attr(S) \text{ or}$$
$$attr(S) \cap attr(T) \rightarrow attr(T)$$

i.e. the common attributes of S and T form a superkey of either S or T.

Dependency Preserving Decomposition

If we can check the functional dependencies of R ***without joining*** S and T then the decomposition is said to be dependency preserving.

Boyce-Codd Normal Form (BCNF)

A relation R is in BCNF, if and only if, for all non-trivial FDs (incl. derived FDs) of the relation, the LHS of every FD is a superkey (i.e. contains a key).

Example: Is the movies relation below in BCNF if we have the following FD: title year → length genre studio?

title	year	length	genre	studio	actor
Star Wars	1977	124	SF	Fox	Carrie Fisher
Star Wars	1977	124	SF	Fox	Mark Hamill
Star Wars	1977	124	SF	Fox	Harrison Ford
Gone with the Wind	1939	231	Drama	MGM	Vivien Leigh
Wayne's World	1992	95	Comedy	Paramount	Dana Carvey
Wayne's World	1992	95	Comedy	Paramount	Mike Myers

Decomposition into BCNF

The following algorithm allows us to systematically decompose a relation into BCNF. The basic idea is to use violating FDs (where the LHS is not a superkey) to guide the decomposition strategy, replacing a relation with a violating FD with 2 new relations, one with all the attributes of the FD, and one with all the attributes of the relation minus the RHS of the FD, effectively allowing us to join the relations when required.

initialise decompositions with R
while ViolatingRelation V in decompositions do {
    let LHS → RHS be a nontrivialFD for V such that
        (i) LHS → attr(V) does not hold for FDset$^+$ of R
        (ii) LHS ∩ RHS={}
    remove V from decompositions
    add relation(LHS ∪ RHS) to decompositions
    add relation(attr(V)-RHS) to decompositions }

i.e. LHS not a superkey of R

Note + i.e. we may need to check a derived FD for decomposed relations

Alternatively we can check if a decomposed relation S is in BCNF, by checking that for every subset α of attr(S) - that {α}$^+$=attr(S) or has no attributes of attr(S)-α

Decomposition into BCNF

Summarising, in BCNF decomposition we take one of the violating FDs, which could be a derived FD, and split the relation, into two child relations.

Then for each of the child relations we independently determine their FDs. A child FD set is the subset of the parent FD set that uses only the child's attributes.

Once this is done, we apply the BCNF procedure to each child, calculating keys for the child relation and checking whether any of the child FDs (including derived child FDs) violate the FDs for the child relation.

The decomposition proceeds recursively.

Note: Because there is sometimes a choice of which violating FD to use to split a relation, **there can be multiple BCNF decompositions that are valid**.

Third Normal Form (3NF)

A relation R is in 3NF, if and only if, the LHS of every nontrivial FD is a superkey (the BCNF test) or if every attribute on the RHS of a FD is prime.

An attribute is **prime** if it is a member of **any key** of the relation. Each attribute on the RHS could be a member of a different key.

Consider the previous example:

R(A,B,C)

A, B → C
C → B

Keys: {A,B} {A,C}

Although A → C violates BCNF it does not violate 3NF because C is prime (C is a member of the key {A, C}

Decomposition into 3NF

Given a relation R and a set of FDs F for R, we can decompose R into a set of decomposed relations D (each of which is in 3NF) as follows:

Let C **be a canonical cover for** F **(i.e. a minimal FD set for R)**

Initialise the set of decomposed relations D **to {}**

Foreach FD: LHS → RHS in C
add a new relation(LHS ∪ RHS) to the set of decomposed relations D

Foreach relation R in D that is a subset of another relation in D
remove R from D

If none of the relations in D includes a key for R
add a new relation(key) to D

Example

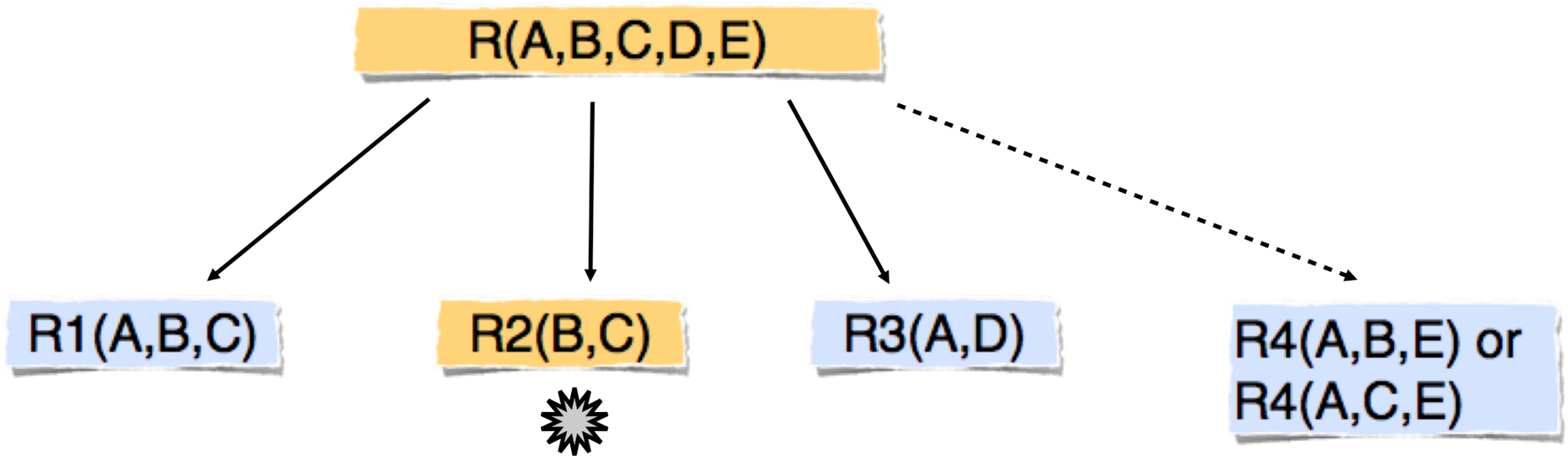
Decompose R(A,B,C,D,E)
into 3NF

FD Set

$A\ B \rightarrow C$
$C \rightarrow B$
$A \rightarrow D$

Keys

{A, B, E}
{A, C, E}



SQL

Nulls

We need to understand the implications of **null**s on arithmetic and comparisons including for joins (see later).

Arithmetic	Any arithmetic that involves a **null** will result in a **null**. Note: In SQL, 0*y where y is **null** is **null**! y-y is also **null** if y is **null**!
Comparisons	Any comparison involving a **null** will result in **unknown**, e.g. x>y where y is **null** will result in **unknown**. **null** is not a constant value like **true** and can't be used in comparisons. To test if an attribute y is null, use y **is null**, or y **is not null** **null** will never match any other value (even **null** itself), unless we explicitly use **is null** or **is not null**.

Scalar Subquery

A select that produces a single value. Scalar subqueries can be used in any expression, e.g. in projection lists, in **where** and **having** clauses. Are often selects with a single aggregate function.

Example: movie(title, year, length, genre, studio, producer)
casting(title, year, name)

```
select title,
       (select count(name)
        from   casting
        where  casting.title=movie.title) as numactors
from   movie;
```

Note the use of a relation from an outer query in the subquery. This is an example of a **correlated subquery** that has to be evaluated multiple times for each outer tuple.

Set Membership Subqueries

Subqueries that produce a set of values can be used to test if a value is a member of the set by using the in or not in operators.

Example: movie(title, year, length, genre, studio, producer)
casting(title, year, name)
studio(name, address, boss)

```
select title
from   movie
where  studio in (select name from studio
   where address like 'C%')
```

We can extend the approach to tuple values enclosed in parentheses:

```
select name
from   casting
where  (title, year) not in
     (select title, year from movie where genre='sf')
```

Relation subqueries

The **exists** and **not exists** functions with a subquery argument can be used to test whether a relation is empty (has no tuples) or not.

The **not unique** function can be used to test whether a relation has duplicates, or hasn't duplicates with **unique**.

Example: movie(title, year, length, genre, studio, producer)
casting(title, year, name)
studio(name, address, boss)

```
select title
from   movie m1
where  not exists (select * from movie m2
                   where m2.title=m1.title and m2.year<>m1.year
)
```

Transactions

Isolation Level Anomalies

ISOLATION LEVEL	Dirty Reads	Non-Repeatable Reads	Phantom Reads
Read Uncommitted	***Possible***	***Possible***	***Possible***
Read Committed	Prevented	***Possible***	***Possible***
Repeatable Read	Prevented	Prevented	***Possible***
Serializable	Prevented	Prevented	Prevented

Dirty writes are not allowed by any Isolation level, i.e. we cannot update data that has been updated by another transaction and has not yet been committed or aborted.

Dirty Read

name	bal
Alice	10
Bob	20

During a transaction a tuple is read twice but attribute values differ.

Start Transaction 1	Start Transaction 2
select bal **from** accounts **where** name='Alice'	**update** accounts **set** bal=99 **where** name='Alice'
select bal **from** accounts **where** name='Alice'	**rollback**

Serializable	Repeatable Read	Read Committed	Read Uncommitted
Prevented	Prevented	Prevented	***Possible***

In Serializable, Repeatable Read and Read Committed, the 2nd select must return (Alice,10), so the DBMS will lock Alice (the tuple) and cause the update to block until transaction 1 commits. For Read Uncommitted, the 2nd select will return (Alice,99). However transaction 2 could subsequently rollback!

Non-Repeatable Read

name	bal
Alice	10
Bob	20

During a transaction a tuple is read twice but attribute values differ.

Start Transaction 1	Start Transaction 2
select * **from** accounts **where** name='Alice' **select** * **from** accounts **where** name='Alice' **commit**	**update** accounts **set** bal=99 **where** name='Alice' **commit**

Serializable	Repeatable Read	Read Committed	Read Uncommitted
Prevented	Prevented	***Possible***	***Possible***

In Serializable and Repeatable Read, the 2nd **select** must return (Alice,10), so will lock Alice and cause the **update** to block until transaction 1 commits. For the other isolation levels, the 2nd **select** will return (Alice,99).

Phantom Read

name	bal
Alice	10
Bob	20

Special case of non-repeatable reads. Tuples returned by a second identical query differ from tuples returned from first query.

Start Transaction 1	Start Transaction 2
select * **from** accounts **where** bal>0 **and** bal<100 **select** * **from** accounts **where** bal>0 **and** bal<100	**insert into** accounts **set** values(Tim, 66) **commit**

Serializable	Repeatable Read	Read Committed	Read Uncommitted
Prevented	***Possible***	***Possible***	***Possible***

In Serializable, the 1st **select** will lock all accounts with balances between 1 and 99 and cause the **insert** to block until transaction 1 commits. For the other isolation levels, **insert** will not block and the 2nd **select** will include (Tim, 66).

Indexing

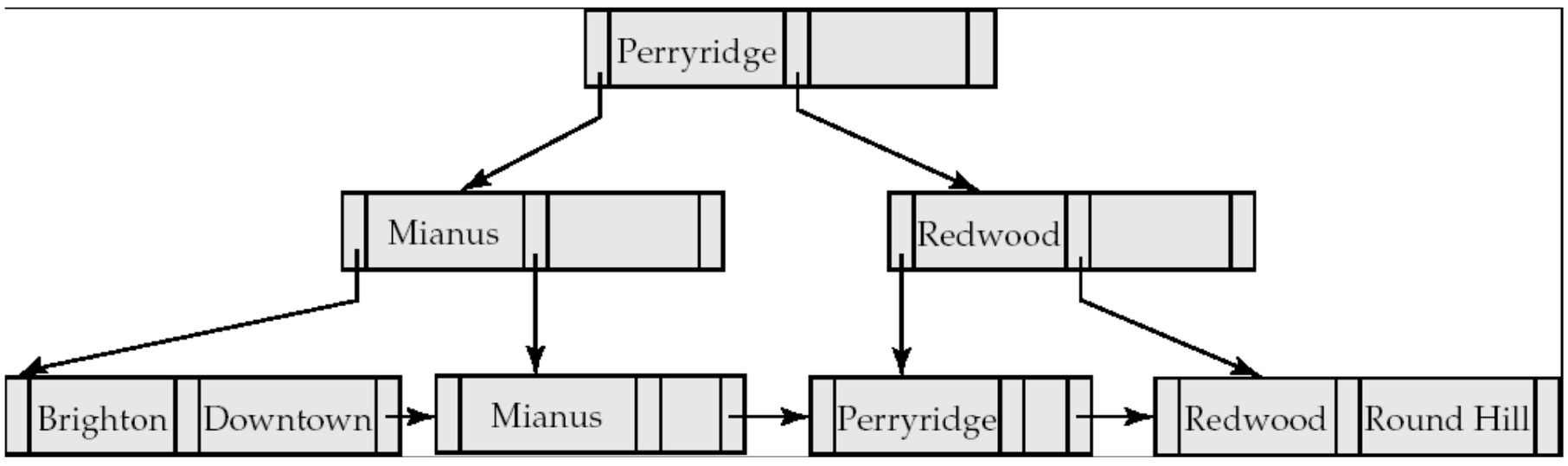
Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute to set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".

B^+-Tree Index Files (Cont.)

- A B^+-tree is a rooted tree satisfying the following properties:
- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
- A leaf node has between $\lceil (n–1)/2 \rceil$ and $n–1$ values
- Special cases:
  - If the root is not a leaf, it has at least 2 children.
  - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n–1)$ values.



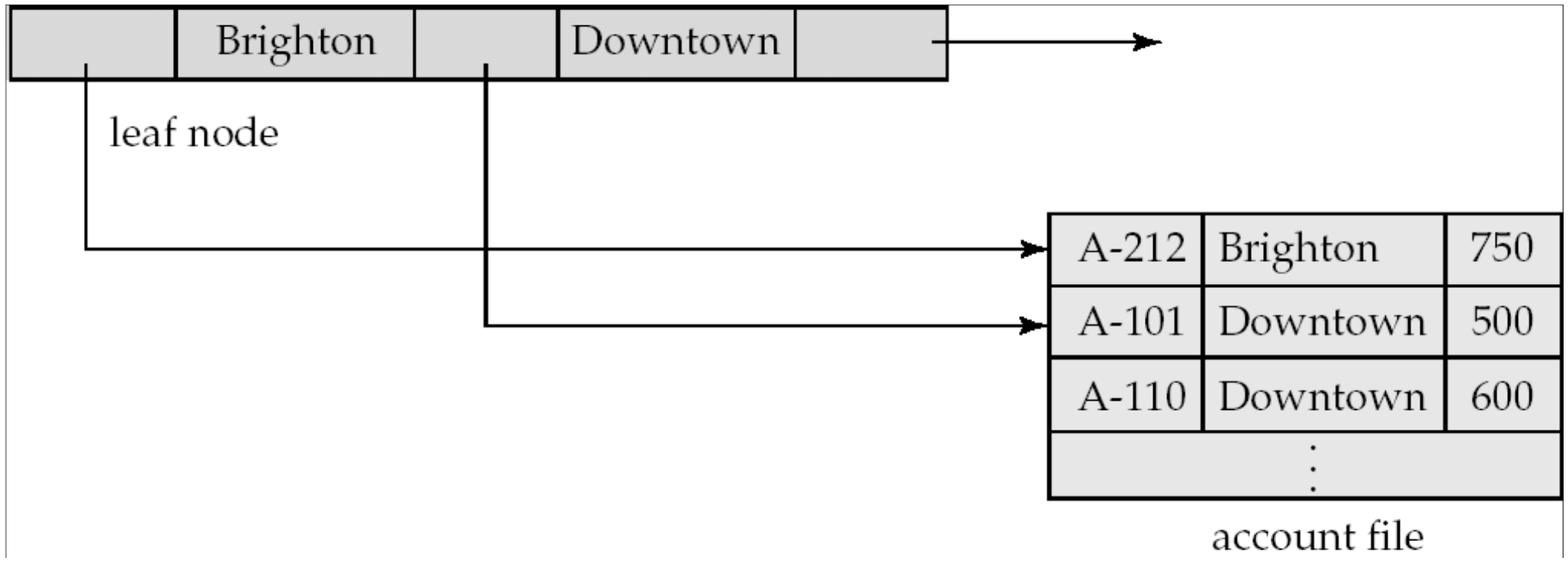
B^+-Tree Node Structure

- Typical node

  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

Leaf Nodes in B$^+$-Trees

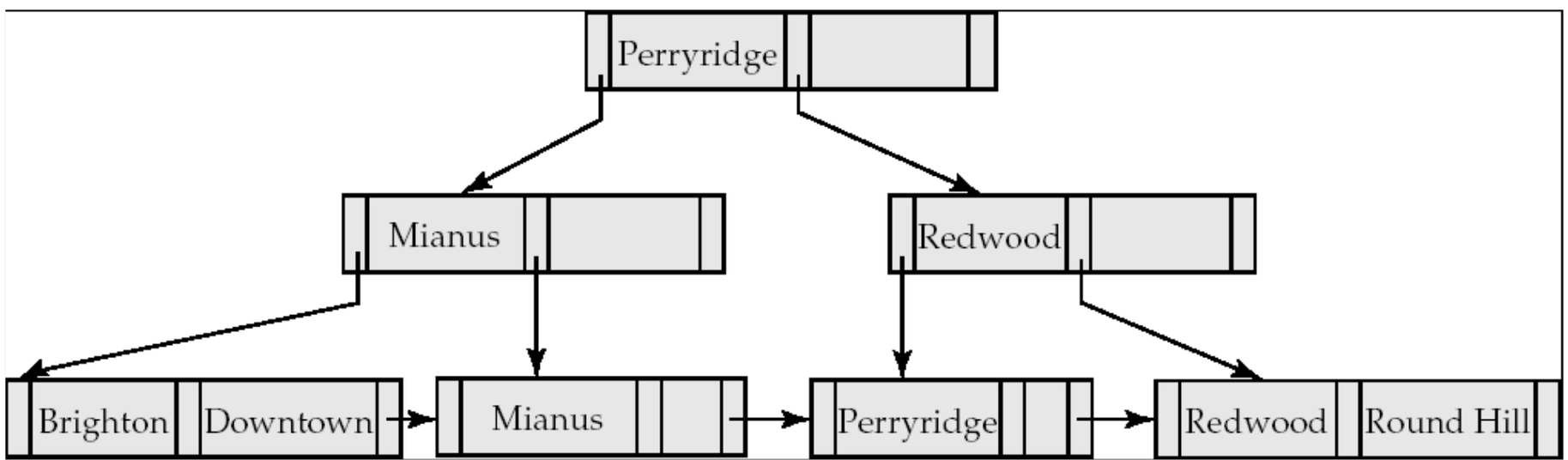


Queries on B$^+$-Trees

Find all records with a search-key value of *k*.

1. *N=root*
2. Repeat
    1. Examine *N* for the smallest search-key value > *k*.
    2. If such a value exists, assume it is K_i. Then set $N = P_i$
    3. Otherwise $k \geq K_{n-1}$. Set $N = P_n$

    Until *N* is a leaf node
3. If for some *i*, key $K_i = k$ follow pointer P_i to the desired record or bucket.
4. Else no record with search-key value *k* exists.



Updates on B$^+$-Trees: Insertion

Find the leaf node in which the search-key value would appear

If the search-key value is already present in the leaf node

1. Add record to the file

If the search-key value is not present, then

1. Add the record to the main file (and create a bucket if necessary)
2. If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
3. Otherwise, split the node (along with the new (key-value, pointer) entry) as discussed in the next slide.

Updates on B+-Trees: Insertion (Cont.)

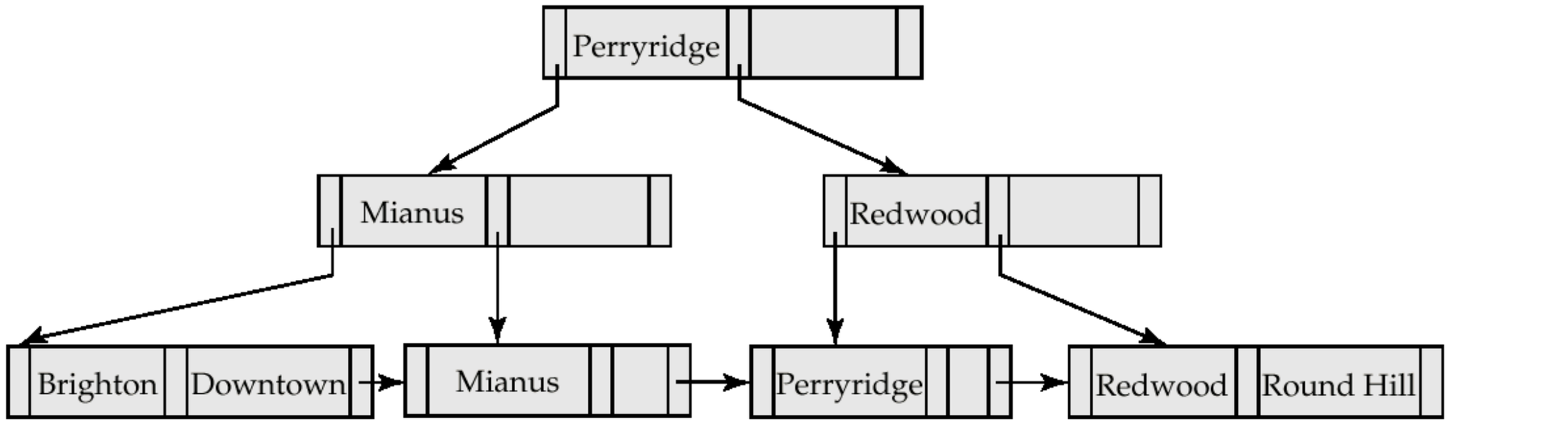
Splitting a leaf node:

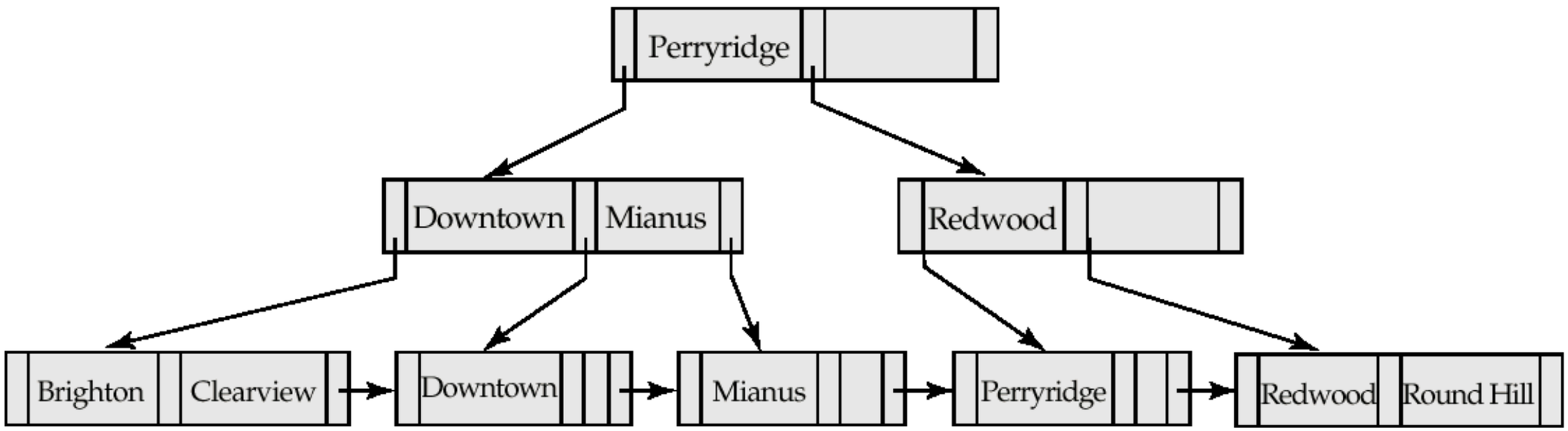
- take the *n* (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first in the original node, and the rest in a new node.
- let the new node be *p,* and let *k* be the least key value in *p.* Insert (*k,p*) in the parent of the node being split.
  - If the parent is full, split it and **propagate** the split further up.

Splitting of nodes proceeds upwards till a node that is not full is found.

- In the worst case the root node may be split increasing the height of the tree by 1.

Updates on B+-Trees: Insertion (Cont.)





B+-Tree before and after insertion of “Clearview”

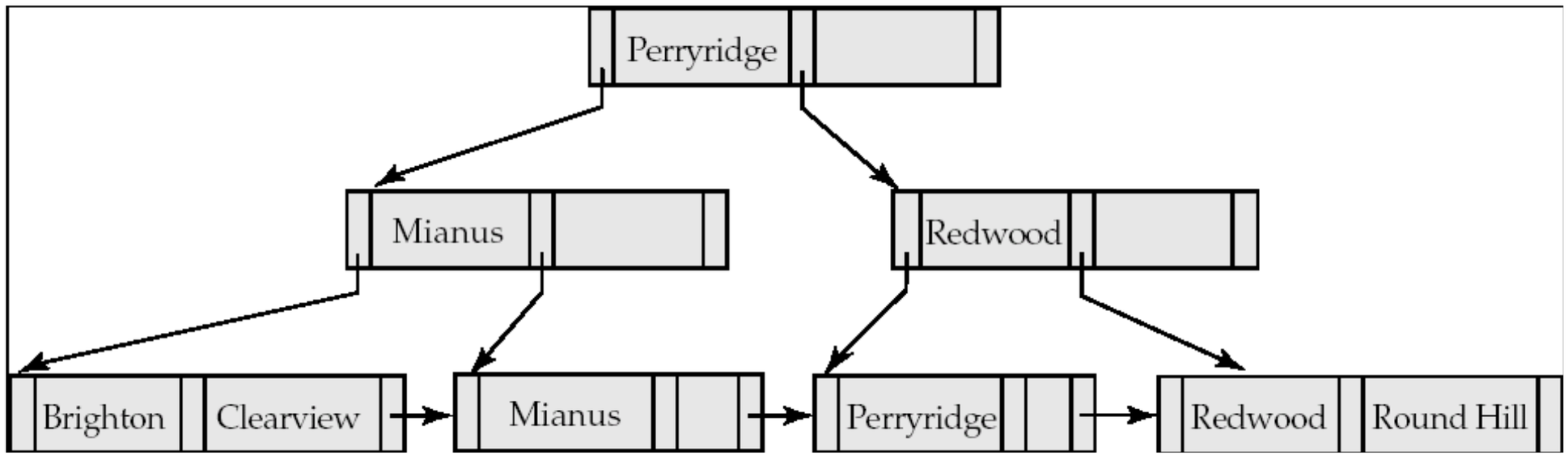
Updates on B$^+$-Trees: Deletion

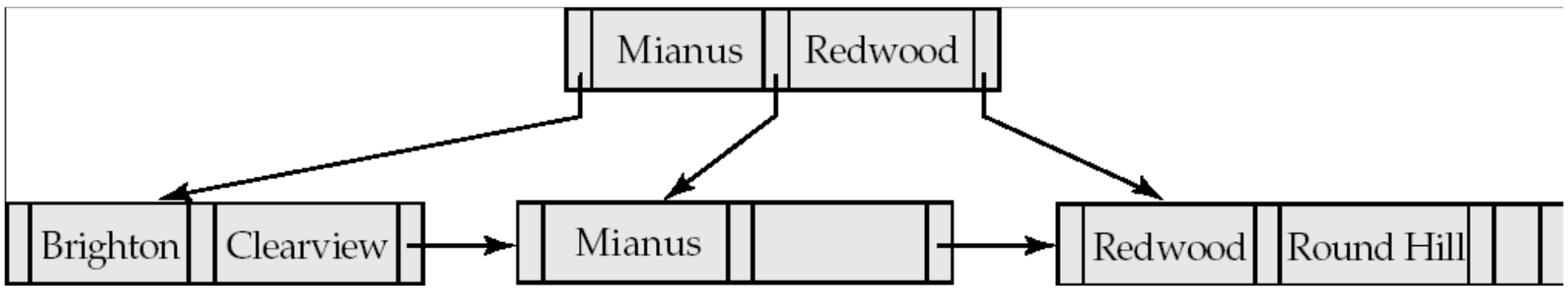
- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then ***merge siblings:***
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.
  - Delete the pair (K_{i-1}, P_i), where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure.

Updates on B$^+$-Trees: Deletion

- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then **redistribute pointers**:
  - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
  - Update the corresponding search-key value in the parent of the node.
- The node deletions may cascade upwards until a node which several pointers is found.
- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

Examples of B+-Tree Deletion (Cont.)





Before and After deletion of “Perryridge” from result of previous example