

Joins and semijoins in relational algebra and SQL

Dirk Van Gucht¹

¹Indiana University

Joins (Discovering Relationships)

- One of the main purposes of query languages, and more generally search languages, is to **discover relationships between objects in the data(base)**.
- Operations in query languages that can aid in discovering such relationships are called **joins**

Joins (Discovering Relationships)

- In our running example, the database has three kinds of objects, i.e., **students**, **courses**, and **teachers**, and two kinds of relationships, i.e., **Enroll** relationships between students and courses, and **TaughtBy** relationships between courses and teachers.
- Starting from this data, we may then wish to discover each relationship (s, t) between a student s and a teacher t indicating that
 - student s takes **some** course taught by teacher t , or
 - student s takes **all** courses taught by teacher t , or
 - student s takes **two** courses taught by teacher t ,
 - etc.

Semijoins (Discovering objects satisfying properties on the basis of relationships)

- Another main purpose of query languages, and more generally search languages, is to **discover objects in the database that satisfy certain properties**, in particular those that can only be established on the basis of relationships of these objects with other objects in the database.

For example

- Find each student s who is enrolled in a class taught by teacher 'Eric'
 - Find each teacher t who teaches no students who take CS courses
-
- Operations in query languages that can aid in discovering objects with such properties are called **semijoins**

Joins

- Two types of joins
 - Regular joins:¹ these are joins that compute relationships between objects (o_1, o_2) based on tuple-component comparisons, like $t_1.A \theta t_2.B$, where o_1 are components of t_1 and o_2 are components of t_2 .
 - Set joins: these are joins that compute relations between objects based on comparisons between sets of tuples associated with these objects, e.g, a comparison between the set of courses taken by a student and the set of courses taught by a teacher.
- Analogously, there are also two types of semijoins.

¹Regular joins are typically just called joins.

Joins in SQL

- Regular joins: SQL supports regular joins with the special operators

JOIN	or, equivalently, INNER JOIN
NATURAL JOIN	special case of JOIN
CROSS JOIN	cartesian product
	special case of NATURAL JOIN

- Set joins: SQL has only limited support for set joins, but they can be simulated using the predicates

[NOT] EXISTS(…) and [NOT] IN
(SELECT COUNT(1) FROM …) θk

Semijoins in SQL

- Regular semijoins: SQL does not have special operations for semijoins, but they can be simulated with the **NATURAL JOIN** operations, or with the **IN** set predicate
- Set semijoins: SQL does not have set semijoins, but they can be shown to be special cases of set joins and as such can be simulated

Regular joins in Relational Algebra (Introduction)

- Consider the query "Find the sid and age of each student who takes a course and whose name is Ann."
- This query can be expressed with the RA expression

$$\pi_{\text{Student.sid}, \text{age}}(\sigma_{\text{sname}=\text{'Ann'}} \wedge \text{Student.sid} = \text{Enroll.sid})(\text{Student} \times \text{Enroll})$$

- Or, using the notation $S.sid$ instead of $\text{Student}.sid$ and $E.sid$ instead of $\text{Enroll}.sid$, more succinctly

$$\pi_{S.sid, \text{age}}(\sigma_{\text{sname}=\text{'Ann'}} \wedge S.sid = E.sid)(\text{Student} \times \text{Enroll})$$

Regular joins in Relational Algebra (Introduction)

- Consider the query "Find the sid and age of each student who takes a course and whose name is Ann."
- This query can be expressed with the RA expression

$$\pi_{S.sid, \text{age}}(\sigma_{\text{pname}=\text{'Ann'}} \wedge S.sid=E.sid (\text{Student} \times \text{Enroll}))$$

- The subexpression

$$\sigma_{\text{pname}=\text{'Ann'}} \wedge S.sid=E.sid (\text{Student} \times \text{Enroll})$$

is called a **(regular) join** and we use the bowtie notation (\bowtie) to write it as follows:

$$\text{Student} \bowtie_{\text{pname}=\text{'Ann'}} \wedge S.sid=E.sid \text{ Enroll}$$

- With the join operation, we can express the query as follow:

$$\pi_{S.sid, \text{age}}(\text{Student} \bowtie_{\text{pname}=\text{'Ann'}} \wedge S.sid=E.sid \text{ Enroll})$$

Regular join in SQL (Introduction)

- "Find the sid and age of each student who takes a course and whose name is Ann."
- In SQL we can formulate this query as follows:

```
SELECT DISTINCT s.sid, s.age  
FROM Student s JOIN Enroll e  
ON (s.sname = 'Ann' AND s.sid = t.sid)
```

- As compared to the alternative SQL specification

```
SELECT DISTINCT s.sid, s.age  
FROM Student s, Enroll e  
WHERE s.sname = 'Ann' AND s.sid = t.sid
```

Regular joins in the Relational Algebra (General case)

- Let $E_1(A_1, \dots, A_m)$ and $E_2(B_1, \dots, B_n)$ be RA expressions and let C be a condition involving the attributes A_1, \dots, A_m and B_1, \dots, B_n .
- Then

$$E_1 \bowtie_C E_2$$

is called the **join between E_1 and E_2 on condition C** .

- The schema of this join is

$$(E_1.A_1, \dots, E_1.A_m, E_2.B_1, \dots, E_2.B_n)$$

- $E_1 \bowtie_C E_2$ is defined to be equivalent with the RA expression

$$\sigma_C(E_1 \times E_2)$$

Regular joins in SQL (General case)

- If Q_{E_1} and Q_{E_2} denote the SQL queries corresponding to E_1 and E_2 then $E_1 \bowtie_C E_2$ corresponds to the SQL query

```
SELECT  e1.A1, ..., e1.Am, e2.B1, ..., e2.Bn
FROM    (QE1) e1 JOIN (QE2) e2 ON (C)
```

- If there is no overlap between the schemas of Q_{E_1} and Q_{E_2} , the query can also be formulated as follows:

```
SELECT  A1, ..., Am, B1, ..., Bn
FROM    (QE1) JOIN (QE2) ON (C)
```

More examples

- "Find the sids of students who take at least two courses."
- In RA

$$\pi_{E.sid}(Enroll \bowtie_{E.sid=E_1.sid \wedge E.cno \neq E_1.cno} Enroll_1)$$

- In SQL,

```
SELECT DISTINCT e.sid
FROM Enroll e JOIN Enroll e1
    ON (e.sid = e1.sid  $\wedge$  e.cno  $\neq$  e1.cno)
```

More examples

- "Find the pids of persons who are not the youngest."
- In RA,

$$\pi_{P.pid}(Person \bowtie_{P.age < P_1.age} Person_1)$$

- In SQL,

```
SELECT DISTINCT p.pid  
FROM Person p JOIN Person p1 ON (p.age < p1.age)
```

Special case: cartesian product \times

- Let $E_1(A_1, \dots, A_m)$ and $E_2(B_1, \dots, B_n)$ be RA expressions
- Then,

$$E_1 \bowtie_{\text{true}} E_2 = E_1 \times E_2$$

- In SQL,

```
SELECT A1, ..., Am, B1, ..., Bn
FROM (QE1) JOIN (QE2) ON (true)
```

- Alternatively, in SQL, with CROSS JOIN

```
SELECT A1, ..., Am, B1, ..., Bn
FROM (QE1) CROSS JOIN (QE2)
```

Special case: Natural joins (Introduction)

- A natural join between two relations is a series of "equality" joins on the common attributes of the relations, followed by a projection to remove redundant columns.
- Consider the relations $\text{Enroll}(\text{sid}, \text{cno}, \text{grade})$ and $\text{TaughtBy}(\text{cno}, \text{tid})$; notice that cno is a common attribute
- To perform the natural join between these relations, we write $\text{Enroll} \bowtie \text{TaughtBy}$, where

$\text{Enroll} \bowtie \text{TaughtBy} =$

$\pi_{E.\text{sid}, E.\text{cno}, E.\text{grade}, T.\text{tid}}(\text{Enroll} \bowtie_{E.\text{cno}=T.\text{cno}} \text{TaughtBy})$

- In SQL,

```
SELECT sid, cno, grade, tid  
FROM   Enroll NATURAL JOIN TaughtBy
```

Multiple Natural joins

- Recall relations $Enroll(sid, cno, grade)$,
 $Student(sid, sname, age)$, and $Course(cno, cname, dept)$.
- "Find the sid and sname of each student who takes a CS course."

$$\pi_{sid, sname}(Student \bowtie Enroll \bowtie \sigma_{dept='CS'}(Course))$$

- In SQL, this can be formulated as follows

```
WITH      CS_Course AS  
        (SELECT * FROM Course WHERE dept = 'CS')  
SELECT    DISTINCT sid, sname  
FROM     Student NATURAL JOIN Enroll  
          NATURAL JOIN CS_Course
```

Natural join (General case)

- Let $E_1(A_1, \dots, A_m, C_1, \dots, C_k)$ and $E_2(C_1, \dots, C_k, B_1, \dots, B_n)$ be RA expressions
- Then

$$E_1 \bowtie E_2$$

is a **natural join** expression with schema $(A_1, \dots, A_m, C_1, \dots, C_k, B_1, \dots, B_n)$ and is defined as the RA expression

$$\pi_{A_1, \dots, A_m, E_1.C_1, \dots, E_1.C_k, B_1, \dots, B_n}(E_1 \bowtie_C E_2)$$

where C is the condition

$$E_1.C_1 = E_2.C_1 \wedge \dots \wedge E_1.C_k = E_2.C_k$$

- Notice that if there are no common C attributes then $E_1 \bowtie E_2 = E_1 \times E_2$; an empty conjunction is interpreted as true

Natural joins (General case) in SQL

- If Q_{E_1} and Q_{E_2} denote the SQL queries corresponding to E_1 and E_2 then $E_1 \bowtie E_2$ corresponds to the SQL query

```
SELECT A1, ..., Am, C1, ..., Ck, B1, ..., Bn
FROM   (QE1) e1 NATURAL JOIN (QE2) e2
```

SQL as an algebra

- SQL is a declarative language.
- However, with the addition of the JOIN, NATURAL JOIN, and CROSS JOIN, **SQL can be seen as an algebra** that can faithfully simulate RA
- Indeed,
 - a WHERE clause can simulate a selection operation σ ;
 - a SELECT DISTINCT clause can simulate a projection operation π ;
 - a CROSS JOIN in the FROM clause can simulate a cartesian product; and
 - SQL has the UNION, INTERSECT, and EXCEPT operator that simulate \cup , \cap , and $-$

Some observations about the power of the join operator

- Let $E(A_1, \dots, A_m)$, $E_1(A_1, \dots, A_m)$, and $E_2(A_1, \dots, A_m)$ be RA expressions with the same schema (A_1, \dots, A_m)
- Then

$$\begin{aligned} E_1 \cap E_2 &= E_1 \bowtie E_2 \\ \sigma_{A_i \theta \mathbf{a}}(E) &= \pi_{E.*}(E \bowtie_{E.A_i \theta B} (B : \mathbf{a})) \\ \sigma_{A_i = A_j}(E) &= \pi_{E.*}(E \bowtie_{E.A_i = E.A_j} E') \end{aligned}$$

where E' is a copy of E .

- Conclusion: if we had taken \bowtie_C as a basic operation in RA, then \times , \cap , and σ become derived operations.
- So all we need are the operations π , \bowtie_C , \cup , and $-$.
- In SQL, this means that we don't need the WHERE clause!

Semijoins (Introduction)

- "Find the student information for each student who takes a course."

$$\pi_{S.sid, S.sname, S.age}(Student \bowtie Enroll)$$

- In this join expression, we "select" those tuples in the Student relation that **join with some** tuple in the Enroll relation.
- However, we are not interested in retaining any information (such as cno and grade) from this Enroll tuple.²
- This operation is called a **semijoin** (denoted \bowtie) and we simply write

Student \bowtie Enroll

²Observe that $\pi_{S.sid, S.sname, S.age}(Student \bowtie Enroll) \subseteq Student$

Semijoin (Introduction)

Observe that

$$\pi_{S.sid, S.sname, S.age}(Student \bowtie Enroll) = Student \bowtie \pi_{E.sid}(Enroll)$$

- Notice that in the right side of this equation all the unnecessary information from the Enroll relation is projected out. Indeed, only its sid column is relevant in this join.
- We could have therefore also defined the semijoin as follows:

$$Student \ltimes Enroll = Student \bowtie \pi_{E.sid}(Enroll)$$

Semijoins in SQL

- SQL does not have a semijoin operator. However semijoins can be simulated in various ways.
- (1) Using the **NATURAL JOIN** operator

```
SELECT sid, sname, age
FROM Student NATURAL JOIN
      (SELECT DISTINCT sid FROM Enroll) q
```

- (2) Using the **IN** predicate

```
SELECT sid, sname, age
FROM Student
WHERE sid IN (SELECT DISTINCT sid FROM Enroll)
```

Semijoin (example)

- "Find the student information of students who take a course in the CS department."

$$\pi_{S.sid, S.sname, S.age}(Student \bowtie Enroll \bowtie \sigma_{C.Dept='CS'}(Course))$$

- With semijoins, this becomes

$$Student \ltimes (Enroll \ltimes \sigma_{C.Dept='CS'}(Course))$$

This expression can be evaluated in linear time using hashing!

- Be careful, **semijoin is not associative**:

$$Student \ltimes (Enroll \ltimes \sigma_{C.Dept='CS'}(Course)) \not\approx (Student \ltimes Enroll) \ltimes \sigma_{C.Dept='CS'}(Course)$$

Semijoin (not associative)

- Be careful, **semijoin is not associative**:

$$Student \ltimes (Enroll \ltimes \sigma_{C.Dept='CS'}(Course)) \not\Leftarrow (Student \ltimes Enroll) \ltimes \sigma_{C.Dept='CS'}(Course)$$

- In fact,

$$(Student \ltimes Enroll) \ltimes \sigma_{C.Dept='CS'}(Course) \Leftrightarrow Student \ltimes Enroll$$

if $\sigma_{C.dept='CS'}(Course) \neq \emptyset$.

Semijoin (General case)

- Let $E_1(A_1, \dots, A_m, C_1, \dots, C_k)$ and $E_2(C_1, \dots, C_k, B_1, \dots, B_n)$ be RA expressions
- Then

$$E_1 \ltimes E_2$$

is the **semijoin** between E_1 and E_2 (with schema $(A_1, \dots, A_m, C_1, \dots, C_k)$) and it is defined as the following RA expression:

$$E_1 \bowtie \pi_{E_2.C_1, \dots, E_2.C_k}(E_2)$$

- $E_1 \ltimes E_2$ returns each tuples e_1 from E_1 wherefore there exists a tuples e_2 of E_2 such that $e_1.C_1 = e_2.C_1 \wedge \dots \wedge e_1.C_k = e_2.C_k$.
- So notice that one always has $E_1 \ltimes E_2 \subseteq E_1$.

Semijoins (General case) in SQL

- If Q_{E_1} and Q_{E_2} denote the SQL queries corresponding to E_1 and E_2 then $E_1 \ltimes E_2$ corresponds to the SQL query

```
SELECT e1.*
FROM (QE1) e1 NATURAL JOIN
      (SELECT DISTINCT C1, ..., Ck FROM (QE2) e2) q
```

- Alternatively, $E_1 \ltimes E_2$ corresponds to the SQL query

```
SELECT e1.*
FROM (QE1) e1
WHERE (e1.C1, ..., e1.Ck) IN
      (SELECT DISTINCT e2.C1, ..., e2.Ck FROM (QE2)
```

Semijoin (Time complexity)

- In general, $E_1 \ltimes E_2$ can be implemented (using hashing) in linear time $O(|E_1| + |E_2|)$.
- By contrast, $E_1 \bowtie E_2$ runs in the worst case in $O(|E_1||E_2|)$.
- Intuition by example:
 - Determining if a student takes a CS course can be done fast (linear time)
 - Determining for each student the CS courses taken by that student can be very expensive (quadratic time)
- Lesson: wherever possible, use semijoins

Anti-semijoins

- Consider the query "Find the sid, name, and age of each student who is not enrolled in any course."
- In RA, this query can be formulated as follows

$\text{Student} - \text{Student} \times \text{Enroll}$.

- This is correct since $\text{Student} \times \text{Enroll}$ gives the students who are enrolled in some courses.
- Therefore, if we take the relation Student and subtract from it $\text{Student} \times \text{Enroll}$, we get indeed the students who are not enrolled in any course.

Anti-semijoins

- "Find the sid, name, and age of each student who is not enrolled in any course". In RA,

$$Student - Student \times Enroll.$$

- There is an operation, called the **anti-semijoin**, and denoted by $\overline{\times}$, which is especially designed for such expressions.
- Using the **anti-semijoin** operation $\overline{\times}$, this query can be more succinctly expressed as

$$Student \overline{\times} Enroll.$$

Anti-semijoins

- In general, given expressions E_1 and E_2 , the anti semijoin between E_1 and E_2 , i.e., $E_1 \setminus E_2$ is defined as the expression

$$E_1 - (E_1 \ltimes E_2)$$

- In SQL, the anti-semijoin can be supported using the **NOT IN** set predicate
- Just like the semijoin, the anti-semijoin can be implemented in linear time $O(|E_1| + |E_2|)$ using hashing

Anti-semijoins (General case) in SQL

- If Q_{E_1} and Q_{E_2} denote the SQL queries corresponding to E_1 and E_2 , then $E_1 \setminus E_2$ corresponds to the SQL query

```
SELECT e1.*  
FROM (QE1)e1  
EXCEPT  
SELECT e1.*  
FROM (QE1) e1 NATURAL JOIN  
(SELECT DISTINCT C1,...,Ck FROM (QE2) e2)q
```

- Alternatively, $E_1 \setminus E_2$ corresponds to the SQL query

```
SELECT e1.*  
FROM (QE1) e1  
WHERE (C1,...,Ck) NOT IN  
(SELECT DISTINCT C1,...,Ck FROM (QE2) e2)
```

Semijoin and Anti-semijoins Special Cases

- Let E_1 and E_2 be expressions with the **same schemas**
- Then

$$E_1 \ltimes E_2 = E_1 \bowtie E_2 = E_1 \cap E_2$$

$$E_1 \overline{\ltimes} E_2 = E_1 - (E_1 \cap E_2) = E_1 - E_2$$

The Relational Algebra

Dirk Van Gucht¹

¹Indiana University

February 5, 2019

Outline

- Motivation
- Syntax of Relational Algebra (RA)
- Semantics of RA in SQL
- Expressing queries in RA

Relational Algebra (Motivation)

- SQL is a declarative language to specify queries over relational databases
- RA is a procedural language wherein algebraic expressions specify queries
- RA expressions provide procedures (algorithms) to evaluate queries
 - Examining these procedure provides insights about the time and space complexities of evaluating queries

Relational Algebra (Motivation)

- SQL¹ and RA express the same queries:
 - Each SQL query can be expressed by an RA expression
 - Each RA expression can be expressed by a SQL query.
- **Query optimization:** Rewrite rules can transform a RA expression into another equivalent RA expression that is more efficient to evaluate
- **Query evaluation:** Different algorithms and data structures can be associated with RA operations for efficient evaluations

¹without aggregation functions

Syntax of RA

- The relational algebra is a **typed** language of expressions
- Starting from relations and constants, RA expressions are inductively built using the operators

Operator	Algebraic notation
cartesian product	\times
selection	$\sigma(\cdot)$
projection	$\pi \dots (\cdot)$
union	\cup
intersection	\cap
difference	$-$

- Each RA expression has a **schema** which is its type
- We will use $E(A_1, \dots, A_m)$ to denote a RA expression E with attribute schema (A_1, \dots, A_m)

RA basic expressions: relations

- Let $R(A_1, \dots, A_m)$ be a relation
- Then R is a RA expression with schema (A_1, \dots, A_m)
- Its value is the set of tuples from R , i.e., the relation instance associated with R
- R can be expressed by SQL query

```
SELECT r.A1, ..., r.Am
FROM   R r
```

RA basic expressions: constants

- Let A be an attribute and let a be a constant
- Then $(A:a)$ is a RA expression with schema (A)
- Its value is a unary relation containing the single tuple (a)
- $(A : a)$ can be expressed by SQL query

SELECT a AS A

A
a

RA operators: the cartesian product \times

- Let $R(A_1, \dots, A_m)$ and $S(B_1, \dots, B_n)$ be two relations with **non-overlapping** schemas
- Then $R \times S$ is a RA expression with schema $(A_1, \dots, A_m, B_1, \dots, B_n)$
- Its value is the set of all tuples that can be obtained by **pairing** each tuple r of R with each tuple s of S
- $R \times S$ can be expressed by the SQL query

```
SELECT r.A1, ..., r.Am, s.B1, ..., s.Bn
  FROM Rr, Ss
```

- Observe that $|R \times S| = |R||S|$

RA operators: the cartesian product \times (example)

R		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2
a_3	b_3	c_3

S	
D	
1	
2	

R \times S			
A	B	C	D
a_1	b_1	c_1	1
a_2	b_2	c_2	1
a_3	b_3	c_3	1
a_1	b_1	c_1	2
a_2	b_2	c_2	2
a_3	b_3	c_3	2

RA operators: the cartesian product \times (building search space)

- $R \times S$ provides in **single** relation all the information that is present in R and S
- $R \times S$ can be interpreted as the **search space**
- It provides a mechanism to associate each tuple r of R with each tuple s of S into a tuple (r, s) whose components can be compared
 - I.e., the components $r.A_1, \dots, r.A_m$ of r can be compared with the components $s.B_1, \dots, s.B_n$ of s

RA operators: selection $\sigma_{A_i \theta a}$ (Introduction)

- Let $R(A_1, \dots, A_m)$ be a relation
- Let a be a constant value
- Let θ be one of the comparison operators $=, \neq, <, \leq, >, \geq$
- Then $\sigma_{A_i \theta a}(R)$ is a RA expression with schema (A_1, \dots, A_m)
- Its value consists of the tuples r in R such that $r.A_i \theta a$ is true
- $\sigma_{A_i \theta a}(R)$ is expressed in SQL by

```
SELECT r.A1, ..., r.Am
FROM   R r
WHERE  r.Ai θ a
```

- Observe that $|\sigma_{A_i \theta a}(R)| \leq |R|$

RA operators: selection $\sigma_{A \theta a}$ (Example)

R

A	B	C
a_1	b_1	c_1
a_2	b_2	c_2
a_2	b_1	c_2
a_3	b_3	c_2

$\sigma_{B=b_1}(R)$

A	B	C
a_1	b_1	c_1
a_2	b_1	c_2

$\sigma_{A \neq a_3}(R)$

A	B	C
a_1	b_1	c_1
a_2	b_2	c_2
a_2	b_1	c_2

RA operators: selection $\sigma_{A_i \theta A_j}$ (Introduction)

- Let $R(A_1, \dots, A_m)$ be a relation
- Let θ be one of the comparison operators $=, \neq, <, \leq, >, \geq$
- Then $\sigma_{A_i \theta A_j}(R)$ is a RA expression with schema (A_1, \dots, A_m)
- Its value is the set of tuples r of R such that $r.A_i \theta r.A_j$ is true
- $\sigma_{A_i \theta A_j}(R)$ is expressed in SQL by

```
SELECT  r.A1, ..., r.Am
FROM    R r
WHERE   r.Ai θ r.Aj
```

- Observe that $|\sigma_{A_i \theta A_j}(R)| \leq |R|$

RA operators: selection $\sigma_{A=B}$ (Example)

R

A	B	C
<i>a</i> ₁	<i>b</i> ₁	<i>a</i> ₁
<i>a</i> ₂	<i>b</i> ₂	<i>c</i> ₂
<i>a</i> ₂	<i>b</i> ₃	<i>a</i> ₂

$\sigma_{A=C}(R)$

A	B	C
<i>a</i> ₁	<i>b</i> ₁	<i>a</i> ₁
<i>a</i> ₂	<i>b</i> ₃	<i>a</i> ₂

$\sigma_{A \neq C}(R)$

A	B	C
<i>a</i> ₂	<i>b</i> ₂	<i>c</i> ₂

RA operators: selection σ

- A selection acts as a **filter** on R
- It provides a **horizontal slice** of R

RA operators: projection π (Introduction)

- Let $R(A_1, \dots, A_m)$ be a relation and let (B_1, \dots, B_k) be a non-empty list of k attributes of R
- Then $\pi_{B_1, \dots, B_k}(R)$ is a RA expression with schema (B_1, \dots, B_k)
- Its value is the relation $\{(r.B_1, \dots, r.B_k) \mid r \in R\}$
- In other words, each tuple r of R is "projected" on its (B_1, \dots, B_k) components.
- $\pi_{B_1, \dots, B_k}(R)$ is expressed in SQL by

```
SELECT DISTINCT    r.B1, ..., r.Bk
  FROM              R r
```

- Observe that $|\pi_{B_1, \dots, B_k}(R)| \leq |R|$

RA operators: projection π (Example)

R

A	B	C
a_1	b_1	c_1
a_2	b_2	c_2
a_1	b_3	c_1
a_2	b_2	c_1

$\pi_{A,C}(R)$

A	C
a_1	c_1
a_2	c_2
a_2	c_1

$\pi_{B,A}(R)$

B	A
b_1	a_1
b_2	a_2
b_3	a_1

Notice that duplicates are eliminated and that projection permits attributes to be permuted

RA operators: projection π

- A projection provides a mechanism to get certain columns from R
- It provides a **vertical slice** of R

Example query in RA and SQL

- Consider the relations Student(sid,sname,age) and Enroll(sno,cno)⁴
- "Find the sid and age of each enrolled student whose name is Ann."
- This query can be expressed by the RA expression
$$\pi_{\text{sid}, \text{age}}(\sigma_{\text{sid}=\text{sno}}(\sigma_{\text{sname}=\text{'Ann'}}(\text{Student} \times \text{Enroll})))$$

- In SQL

```
SELECT DISTINCT s.sid, s.age
FROM Student s, Enroll e
WHERE s.sname = 'Ann' AND s.sid = e.sno
```

⁴Notice that we have used two different names sid and sno to refer uniquely to a student. We did this to ensure that the relations have no commonly named attributes.

Boolean conditions in selection operations

- It is possible to extend the selection operations by permitting boolean combinations of basic conditions " $A_j \theta a$ " and " $A_i \theta A_j$ " using the boolean connectors \wedge , \vee , and \neg .
- "Find the sid and age of each enrolled student whose name is Ann."
- This query can then be expressed using the following RA expression

$$\pi_{\text{sid}, \text{age}}(\sigma_{\text{sname} = \text{'Ann'}} \wedge \text{sid} = \text{sno}(\text{Student} \times \text{Enroll}))$$

- Notice that the structure of this query is nearly the same as that of its SQL equivalent

Some comments about attribute names (renaming)

- Cartesian products require that the attributes names of the participating expressions do not overlap
- There are various ways to overcome this by renaming attributes or using some conventions
- For example, let $R(A, B)$ and $S(B, C)$ be two relations. We can not write $R \times S$ since attribute B occurs in the schemas of R and S . Nonetheless we will permit this and agree that the output schema of $R \times S$ is $(A, R.B, S.B, C)$ where we have used the relation names R and S to differentiate the B attribute in R from the B attribute in S

Some comments about attribute names

- We can also not write $R \times R$. To overcome this problem, we assume that for each relation name R , there is a series of relations R_1, R_2, \dots that are "copies" of R .
- Then instead of writing $R \times R$, we write $R_1 \times R_2$ and the output schema will be $(R_1.A, R_1.B, R_2.A, R_2.B)$
- In SQL, we can always use the **AS** clause to deal with attribute renaming. We will not give the details of attribute renaming in general but assume that it can always be done appropriately.

Some comments about attribute names (Example)

- "Find the sids of students who take at least two courses."

$$\pi_{E1.sid}(\sigma_{E1.sid = E2.sid \wedge E1.cno \neq E2.cno}(Enroll_1 \times Enroll_2))$$

where E_1 and E_2 are abbreviations for $Enroll_1$ and $Enroll_2$, respectively

Boolean set operations (union, intersection, and set difference)

- Let E_1 and E_2 be two RA expressions with the **same** schema then
 - $E_1 \cup E_2$,
 - $E_1 \cap E_2$, and
 - $E_1 - E_2$
- are RA expressions with the same schema
- If Q_{E_1} and Q_{E_2} are the SQL queries corresponding to E_1 and E_2 , respectively, then the expressions are expressed in SQL by
 - $Q_{E_1} \text{ UNION } Q_{E_2}$,
 - $Q_{E_1} \text{ INTERSECT } Q_{E_2}$, and
 - $Q_{E_1} \text{ EXCEPT } Q_{E_2}$

Examples

- In this example, assume relation $Enroll(sid, cno, grade)$
- "Find the sid of each student who received an 'A' in some course."

$$\pi_{sid}(\sigma_{grade='A'}(Enroll))$$

- "Find the sid of each student who received an 'A' in some course and a 'B' in some course."

$$\pi_{sid}(\sigma_{grade='A'}(Enroll)) \cap \pi_{sid}(\sigma_{grade='B'}(Enroll))$$

- "Find the sid of each student who did not receive an 'A' in any course."

$$\pi_{sid}(Student) - \pi_{sid}(\sigma_{grade='A'}(Enroll))$$

- "Find the sids of students who are enrolled in at least one course"

$$\pi_{sid}(Enroll)$$

- "Find the sids of students who take at least two courses"

$$\pi_{E1.sid}(\sigma_{E1.sid = E2.sid \wedge E1.cno \neq E2.cno}(Enroll_1 \times Enroll_2))$$

- "Find the sids of students who take exactly one course"

$$\pi_{sid}(Enroll) - \pi_{E1.sid}(\sigma_{E1.sid = E2.sid \wedge E.cno \neq E2.cno}(Enroll_1 \times Enroll_2))$$

Examples (**NOT ONLY** and **ONLY**)

- "Find the cnos of CS courses."

$$\pi_{cno}(\sigma_{dept='CS}(Course))$$

Denote this expression by *CS*

- "Find sids of students who do **not only** take CS courses."

$$\pi_{sid}(Enroll - (\pi_{sid}(Student) \times CS))$$

- "Find sids of students who **only** take CS courses."

$$\pi_{sid}(Student) - \pi_{sid}(Enroll - (\pi_{sid}(Student) \times CS))$$

Examples (**NOT ALL** and **ALL**)

- "Find sids of students who do **not** take **all** CS courses"

$$\pi_{sid}((\pi_{sid}(Student) \times CS) - Enroll)$$

- "Find sids of students who take **all** CS courses"

$$\pi_{sid}(Student) - \pi_{sid}(\pi_{sid}(Student) \times CS) - Enroll)$$

Examples

Consider the relation $\text{Person}(pid, age)$

- "Find the pids of persons who are not the youngest."

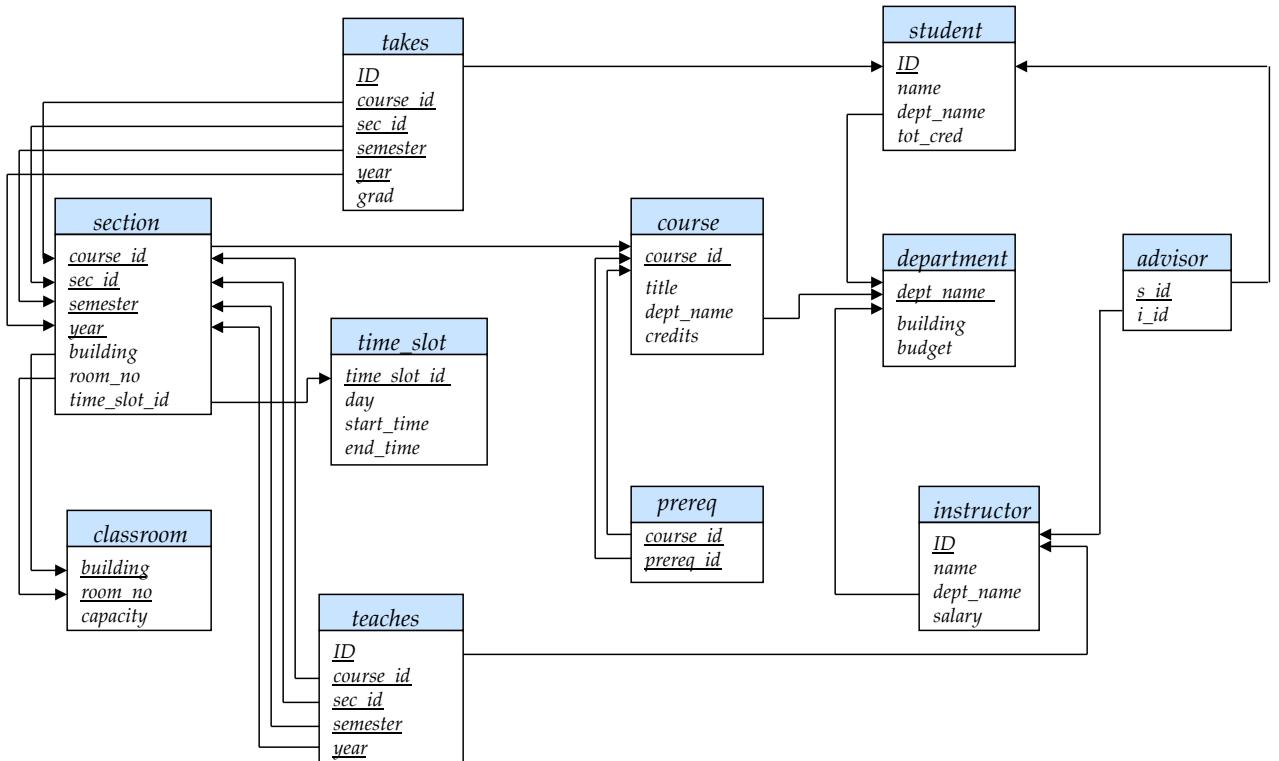
$$\pi_{P1.pid}(\sigma_{P1.age > P2.age}(\text{Person}_1 \times \text{Person}_2))$$

- "Find the pids of the youngest persons."

$$\pi_{pid}(\text{Person}) - \pi_{P1.pid}(\sigma_{P1.age > P2.age}(\text{Person}_1 \times \text{Person}_2))$$

1 Database Schema Diagrams

We will use the university sample relational schema defined in lectures. The following schema diagram illustrates all the relations in the database. Each relation has primary keys underlined. Draw arrows to show all *foreign key* constraints.



2 Relational Algebra

We use relational algebra to specify *queries* on a database. This is the formal mathematical notational. Later we will see how this translates into SQL.

Summary of Relational Algebra

¹Diagrams and examples taken from Database System Concepts 6th Edition

Symbol (Name)	Example of Use
σ (Selection)	$\sigma_{\text{salary} >= 85000}(\text{instructor})$ Return rows of the input relation that satisfy the predicate.
Π (Projection)	$\Pi_{ID, \text{salary}}(\text{instructor})$ Output specified attributes from all rows of the input relation. Remove duplicate tuples from the output.
\bowtie (Natural join)	$\text{instructor} \bowtie \text{department}$ Output pairs of rows from the two input relations that have the same value on all attributes that have the same name.
\times (Cartesian product)	$\text{instructor} \times \text{department}$ Output all pairs of rows from the two input relations (regardless of whether or not they have the same values on common attributes)
\cup (Union)	$\Pi_{name}(\text{instructor}) \cup \Pi_{name}(\text{student})$ Output the union of tuples from the two input relations.

2.1 Lets practice...

The university schema and relations are at the end of this handout.

1. What is the result of $\sigma_{s_id=ID}(\text{student} \times \text{advisor})$?
2. Explain what each of the following expressions do:
 - (a) $\sigma_{year \geq 2009}(\text{takes}) \bowtie \text{student}$
 - (b) $\sigma_{year \geq 2009}(\text{takes} \bowtie \text{student})$
 - (c) $\Pi_{ID, \text{name}, \text{course_id}}(\text{student} \bowtie \text{takes})$
3. Suppose we have the following relational database:

*employee (person_name, street, city)
 works(person_name, company_name, salary)
 company (company_name, city)*

- (a) Find the names of all employees who live in city ‘Miami’.

$$\Pi_{\text{person_name}}(\sigma_{\text{city}='Miami'}(\text{employee}))$$

- (b) Find the names of all employees whose salary is greater than \$100,000.

$$\Pi_{\text{person_name}}(\sigma_{\text{salary} > 100000}(\text{works}))$$

- (c) Find the names of all employees who live in ‘Miami’ and whose salary is greater than \$100,000.

$$\Pi_{person_name}(\sigma_{city='Miami'} \wedge salary > 100000(employee \bowtie works))$$

- (d) Find the names of all employees who work for “First Bank Corporation”.

$$\Pi_{person_name}(\sigma_{company_name='First Bank Corporation'}(works))$$

- (e) Find the names and cities of residence of all employees who work for “First Bank Corporation”.

$$\Pi_{person_name, city}(\sigma_{company_name='First Bank Corporation'}(works \bowtie employee))$$

- (f) Find the names, street address, and cities of residence of all employees who work for “First Bank Corporation” and earn more than \$10,000.

$$\Pi_{person_name, street, city}(\sigma_{company_name='First Bank Corporation'} \wedge salary > 10000(employee \bowtie works))$$

4. Suppose we have the following relational database:

```

branch(branch_name, branch_city, assets)
customer(customer_name, customer_street, customer_city)
loan(loan_number, branch_name, amount)
borrower(customer_name, loan_number)
account(account_number, branch_name, balance)
depositor(customer_name, account_number)

```

- (a) Underline appropriate primary keys.

Primary keys are bold font. Note that some of this is up to interpretation of the database - feel free to discuss and come to some agreement. A sample solution might be:

```

branch(branch_name, branch_city, assets)
customer(customer_name, customer_street, customer_city) ** customer_names may
not be unique**
loan(loan_number, branch_name, amount) ** assume that loan numbers are across all
branches
borrower(customer_name, loan_number)
account(account_number, branch_name, balance)
depositor(customer_name, account_number)

```

- (b) Given your choice of primary keys, identify appropriate foreign keys. Draw a relation diagram to indicate foreign keys.

try having the students draw a box for each schema and then draw the relation diagram for the foreign keys as is done on the diagram on the first page.

$\text{branch}(\text{branch_name}, \text{branch_city}, \text{assets})$
 $\text{customer}(\text{customer_name}, \text{customer_street}, \text{customer_city}) \text{ ** names may not be unique**}$
 $\text{loan}(\text{loan_number}, \text{branch_name}, \text{amount}) \text{ ** assume that loan numbers are across all branches}$
 $\text{borrower}(\text{customer_name}, \text{loan_number})$
 $\text{account}(\text{account_number}, \text{branch_name}, \text{balance})$
 $\text{depositor}(\text{customer_name}, \text{account_number})$

Foreign Keys:

loan and account have foreign key branch_name for branch .

borrower has foreign key loan_number for loan .

depositor has foreign key account_number for account .

- (c) Find the names of all branches located in ‘Chicago’.

$\Pi_{\text{branch_name}}(\sigma_{\text{branch_city}=\text{'Chicago'}}(\text{branch}))$

- (d) Find the names of all borrowers who have a loan in branch ‘Down-town’.

$\Pi_{\text{customer_name}}(\sigma_{\text{branch_name}=\text{'Down-town'}}(\text{borrower} \bowtie \text{loan}))$

- (e) Find all loan numbers with a loan value greater than \$10, 000.

$\Pi_{\text{loan_number}}(\sigma_{\text{amount}>10000}(\text{loan}))$

- (f) Find the names of all depositors who have an account with a value greater than \$6000.

$\Pi_{\text{customer_name}}(\sigma_{\text{balance}>6000}(\text{depositor} \bowtie \text{account}))$

3 University Relations

Relations and their schemas:

$\text{classroom}(\text{building}, \underline{\text{room_number}}, \text{capacity})$
 $\text{department}(\underline{\text{dept_name}}, \text{building}, \text{budget})$
 $\text{course}(\underline{\text{course_id}}, \underline{\text{title}}, \underline{\text{dept_name}}, \text{credits})$
 $\text{instructor}(\underline{\text{ID}}, \underline{\text{name}}, \underline{\text{dept_name}}, \text{salary})$
 $\text{section}(\underline{\text{course_id}}, \underline{\text{sec_id}}, \underline{\text{semester}}, \underline{\text{year}}, \text{building}, \underline{\text{room_number}}, \underline{\text{time_slot_id}})$
 $\text{teaches}(\underline{\text{ID}}, \underline{\text{course_id}}, \underline{\text{sec_id}}, \underline{\text{semester}}, \underline{\text{year}})$
 $\text{student}(\underline{\text{ID}}, \underline{\text{name}}, \underline{\text{dept_name}}, \text{tot_cred})$
 $\text{takes}(\underline{\text{ID}}, \underline{\text{course_id}}, \underline{\text{sec_id}}, \underline{\text{semester}}, \underline{\text{year}}, \text{grade})$
 $\text{advisor}(\underline{s_ID}, \underline{i_ID})$
 $\text{time_slot}(\underline{\text{time_slot_id}}, \underline{\text{day}}, \underline{\text{start_time}}, \underline{\text{end_time}})$
 $\text{prereq}(\underline{\text{course_id}}, \underline{\text{prereq_id}})$

ID	course_id	sec_id	semester	year	ID	name	dept_name	salary
10101	CS-101	1	Fall	2009	10101	Srinivasan	Comp. Sci.	65000
10101	CS-315	1	Spring	2010	12121	Wu	Finance	90000
10101	CS-347	1	Fall	2009	15151	Mozart	Music	40000
12121	FIN-201	1	Spring	2010	22222	Einstein	Physics	95000
15151	MU-199	1	Spring	2010	32343	El Said	History	60000
22222	PHY-101	1	Fall	2009	33456	Gold	Physics	87000
32343	HIS-351	1	Spring	2010	45565	Katz	Comp. Sci.	75000
45565	CS-101	1	Spring	2010	58583	Califieri	History	62000
45565	CS-319	1	Spring	2010	76543	Singh	Finance	80000
76766	BIO-101	1	Summer	2009	76766	Crick	Biology	72000
76766	BIO-301	1	Summer	2010	83821	Brandt	Comp. Sci.	92000
83821	CS-190	1	Spring	2009	98345	Kim	Elec. Eng.	80000

Teaches

Instructor

course_id	sec_id	semester	year	building	room_number	time_slot_id	course_id	prereq_id
BIO-101	1	Summer	2009	Painter	514	B	BIO-301	BIO-101
BIO-301	1	Summer	2010	Painter	514	A	BIO-399	BIO-101
CS-101	1	Fall	2009	Packard	101	H	CS-190	CS-101
CS-101	1	Spring	2010	Packard	101	F	CS-315	CS-101
CS-190	1	Spring	2009	Taylor	3128	E	CS-319	CS-101
CS-190	2	Spring	2009	Taylor	3128	A	CS-347	CS-101
CS-315	1	Spring	2010	Watson	120	D	EE-181	PHY-101
CS-319	1	Spring	2010	Watson	100	B		
CS-319	2	Spring	2010	Taylor	3128	C		
CS-347	1	Fall	2009	Taylor	3128	A		
EE-181	1	Spring	2009	Taylor	3128	C		
FIN-201	1	Spring	2010	Packard	101	B		
HIS-351	1	Spring	2010	Painter	514	C		
MU-199	1	Spring	2010	Packard	101	D		
PHY-101	1	Fall	2009	Watson	100	A		

Section

Prereq

dept_name	building	budget	course_id	title	dept_name	credits
Biology	Watson	90000	BIO-101	Intro. to Biology	Biology	4
Comp. Sci.	Taylor	100000	BIO-301	Genetics	Biology	4
Elec. Eng.	Taylor	85000	BIO-399	Computational Biology	Biology	3
Finance	Painter	120000	CS-101	Intro. to Computer Science	Comp. Sci.	4
History	Painter	50000	CS-190	Game Design	Comp. Sci.	4
Music	Packard	80000	CS-315	Robotics	Comp. Sci.	3
Physics	Watson	70000	CS-319	Image Processing	Comp. Sci.	3
			CS-347	Database System Concepts	Comp. Sci.	3
			EE-181	Intro. to Digital Systems	Elec. Eng.	3
			FIN-201	Investment Banking	Finance	3
			HIS-351	World History	History	3
			MU-199	Music Video Production	Music	3
			PHY-101	Physical Principles	Physics	4

Department

Course

Set Joins and Semijoins in Relational Algebra

Dirk Van Gucht¹

¹Indiana University

Joins and semijoins

- Two types of joins
 - **Regular joins**: these are joins that compute relationships between objects (o_1, o_2) based on **tuple component comparisons** (see the previous lecture on regular joins.)
 - **Set joins**: these are joins that compute relationships between objects (o_1, o_2) based on **set comparisons between sets** $S(o_1)$ and $S(o_2)$ that are associated with objects o_1 and o_2 , respectively.
 - Think for example of finding pairs (s, p) such that student s only takes courses taught by professors.
 - If $S(s)$ denotes the set of courses taken by s and $S(p)$ denotes the set of courses taught by p , then we need to check if $C(s) \subseteq C(p)$.
- Analogously, there are also two types of semijoins: regular semijoins and set semijoins

Set joins (Motivation)

- Consider the relations `Enroll(sid,cno)` and `TaughtBy(cno,pid)`.
- (s, c) is in `Enroll` when student s is enrolled in course c
- (c, p) is in `TaughtBy` when course c is taught by professor p
- We are interested in student-professor relationships (s, p) based on some complex join condition.
- "Find the student-professor pairs (s, p) such that student s takes [some|not only|not all|no|only|all|at least 2|...] course(s) taught by professor p ."

Set joins (Motivation)

- Consider the relations `Patient(pid, symptom)` and `Disease(sympton,disease)`.
- (p, s) is in `Patient` when patient p exhibits health symptom s
- (s, d) is in `Disease` when s is a symptom of disease d
- "Find the patient-disease pairs (p, d) such that patient p has [some|not only|not all|no|only|all|at least 2| \dots] symptoms associated with disease d ."

Set **semijoins** (Motivation)

- Consider the relations $\text{Patient}(\text{pid}, \text{sympton})$ and $\text{Flu}(\text{sympton})$.
- We are interested in finding patient that satisfy some complex condition.
- "Find each patient p such that p has
[*some|not only|not all|no|only|all|at least 2|...|*] flu symptoms.

Set **semijoins** (Advanced Google Search Engine)

- Consider the relations **Document(doc,word)** and the set of words $\{Advanced, Database, Concept\}$.
- A pair $(d, w) \in Document$ when document d contains the word w .
- Advanced Google Search supports the following queries:
"Find each document d such that d contains
 $[some(any)|no|all]$ words in
 $\{Advanced, Database, Concepts\}$.
- You can try this out for the all query and you will see documents that refer to B561.
- https://www.google.com/advanced_search
- So set semijoins are frequently used in real search applications

Set joins and semijoin in RA (Objective)

- Derive relational algebra expressions for set joins and semijoins that supports queries with quantifiers.
- The technique we use for this follows the Venn-diagram method used in the lecture on "Queries with Quantifiers" to derive SQL queries for such queries.
- We will discuss how the derived RA expressions provide deep insight into the time and space complexities for different set joins
- Note: this is a very technical lecture that relies on understanding of logic and set theory

Set joins in RA (Introduction-Terminology)

We begin by introducing some terminology and state our assumptions.

- Denote by \mathcal{S} the set of all student sids
- Denote by \mathcal{P} the set of all professor pids
- Denote by \mathcal{C} the set of all course cnos
- Denote by E the relation *Enroll*
- Denote by T the relation *TaughtBy*
- We will assume that the natural foreign key constraints hold:

$$\begin{aligned}\pi_{sid}(E) &\subseteq \mathcal{S} \\ \pi_{cno}(E) &\subseteq \mathcal{C} \\ \pi_{cno}(T) &\subseteq \mathcal{C} \\ \pi_{pid}(T) &\subseteq \mathcal{P}\end{aligned}$$

Set joins in RA (Parameterized sets)

- Let s be a student sid (i.e., $s \in S$) and let p be a professor pid (i.e., $p \in P$).
- We will consider **sets parameterized** by s and p .
- Both these sets are subsets of C , i.e., the set of cnos of courses.
 - $E(s)$ is the set of cnos of courses in which student s is enrolled.¹

$$E(s) = \{c \mid (s, c) \text{ is a tuple in the Enroll relation}\}$$

- $T(p)$ is the set of cnos of courses taught by professor p .²

$$T(p) = \{c \mid (c, p) \text{ is a tuple in the TaughtBy relation}\}$$

¹If s is not enrolled in any course then $E(s) = \emptyset$.

²If p does not teach any course then $T(p) = \emptyset$.

Set joins (Regions)

- For each student-professor pair (s, p) , the sets $E(s)$ and $T(p)$ define 4 pairwise disjoint "regions" in the Venn diagram associated with these sets.³
- These regions partition the set of courses \mathcal{C} :

$E(s) \cap T(p)$ = the courses **enrolling** student s and
taught by professor p

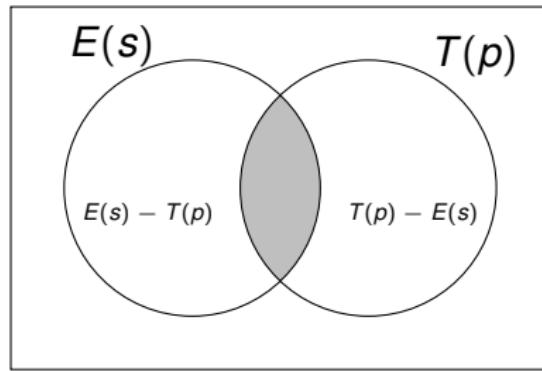
$E(s) - T(p)$ = the courses **enrolling** student s but
not taught by professor p

$T(p) - E(s)$ = the courses **not enrolling** student s but
taught by professor p

$\mathcal{C} - (E(s) \cup T(p))$ = the courses **not enrolling** student s and
not taught by professor p

³It is recommended that you revisit the lecture of "Queries with Quantifiers" to further understand this Venn diagram.

Venn diagram and regions for $E(s)$ and $T(p)$



The gray region is $E(s) \cap T(p)$

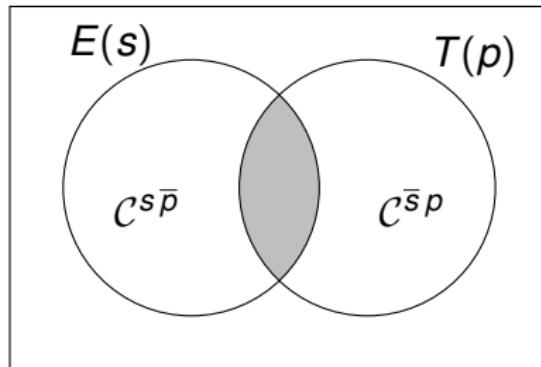
The region outside both $E(s)$ and $T(p)$ is $\mathcal{C} - (E(s) \cup T(p))$

Set joins (Regions)

- For each student-professor pair (s, p) , the sets $E(s)$ and $T(p)$ define 4 pairwise disjoint "regions" that partition the set of cnos of courses \mathcal{C} :
- We use the following notations to denote these regions:

Region	Notation
$E(s) \cap T(p)$	\mathcal{C}^{sp}
$E(s) - T(p)$	$\mathcal{C}^{s\bar{p}}$
$T(p) - E(s)$	$\mathcal{C}^{\bar{s}p}$
$\mathcal{C} - (E(s) \cup T(p))$	$\mathcal{C}^{\bar{s}\bar{p}}$

Venn diagram and regions for $E(s)$ and $T(p)$



The gray region is C^{sp}

The region outside both $E(s)$ and $T(p)$ is $C^{\bar{s}\bar{p}}$. (From now on, we will ignore this region in the lecture.)

Set joins (Region analysis)

- Consider again the regions \mathcal{C}^{sp} , $\mathcal{C}^{s\bar{p}}$, and $\mathcal{C}^{\bar{s}p}$.
- These regions can be either not-empty or empty.
- Depending on this, we can distinguish 6 set joins. They correspond to the **some**, **no**, **not only**, **only**, **not all**, and **all** quantifiers we considered earlier:

$$\mathcal{C}^{sp} \neq \emptyset \leftrightarrow \text{some}$$

$$\mathcal{C}^{sp} = \emptyset \leftrightarrow \text{no}$$

$$\mathcal{C}^{s\bar{p}} \neq \emptyset \leftrightarrow \text{not only}$$

$$\mathcal{C}^{s\bar{p}} = \emptyset \leftrightarrow \text{only}$$

$$\mathcal{C}^{\bar{s}p} \neq \emptyset \leftrightarrow \text{not all}$$

$$\mathcal{C}^{\bar{s}p} = \emptyset \leftrightarrow \text{all}$$

- Once the analysis of these cases is understood, it becomes straightforward to study a very large set of other interesting (and practical) set joins.

Set join (not-empty cases)

- We begin with the set joins for **some**, **not only**, and **not all**. I.e., those corresponding to **non-empty** regions.
- To perform our analysis, it helps to formulate precisely the queries we want to express as RA expressions:

some	$\leftrightarrow \{(s, p) \exists c : c \in \mathcal{C}^{sp}\}$
	$\leftrightarrow \{(s, p) \exists c : c \in E(s) \cap T(p)\}$
	$\leftrightarrow \{(s, p) \exists c : c \in E(s) \wedge c \in T(p)\}$
not only	$\leftrightarrow \{(s, p) \exists c : c \in \mathcal{C}^{s\bar{p}}\}$
	$\leftrightarrow \{(s, p) \exists c : c \in (E(s) - T(p))\}$
	$\leftrightarrow \{(s, p) \exists c : c \in E(s) \wedge c \notin T(p)\}$
not all	$\leftrightarrow \{(s, p) \exists c : c \in \mathcal{C}^{\bar{s}p}\}$
	$\leftrightarrow \{(s, p) \exists c : c \in (T(p) - E(s))\}$
	$\leftrightarrow \{(s, p) \exists c : c \in T(p) \wedge c \notin E(s)\}\}$

Set join (**some** case)

- Recall

$$\begin{aligned}\text{some} &\leftrightarrow \{(s, p) | \exists c : c \in C^{sp}\} \\ &\leftrightarrow \{(s, p) | \exists c : c \in E(s) \wedge c \in T(p)\} \\ &\leftrightarrow \{(s, p) | \exists c : E(s, c) \wedge T(c, p)\}\end{aligned}$$

- We first need to find an RA expression for the set

$$\{(s, c, p) | E(s, c) \wedge T(c, p)\}.$$

- But this is easy since, by the definition of the natural join \bowtie

$$\{(s, c, p) | E(s, c) \wedge T(c, p)\} = E \bowtie T.$$

Consequently, the RA expression for **some** is

$$\pi_{sid,pid}(E \bowtie T)$$

Set join (**not only** case)

- Recall

$$\begin{aligned}\text{not only} &\leftrightarrow \{(s, p) | \exists c : c \in \mathcal{C}^{s \bar{p}}\} \\ &\leftrightarrow \{(s, p) | \exists c : c \in E(s) \wedge \neg(c \in T(p))\} \\ &\leftrightarrow \{(s, p) | \exists c : E(s, c) \wedge \neg T(c, p)\}\end{aligned}$$

- We first need to find an RA expression for the set

$$\{(s, c, p) | E(s, c) \wedge \neg T(c, p)\}.$$

$$\begin{aligned}\{(s, c, p) | E(s, c) \wedge \neg T(c, p)\} &= \{(s, c, p) | (E(s, c) \wedge p \in \mathcal{P}) \wedge \neg(E(s, c) \wedge T(c, p))\} \quad (1) \\ &= \{(s, c, p) | (s, c, p) \in (E \times \mathcal{P}) \wedge \neg((s, c, p) \in E \bowtie T)\} \\ &= (E \times \mathcal{P}) - (E \bowtie T)\end{aligned}$$

The equality (1) will be shown on the next slides.
Consequently, the RA expression for **not only** is

$$\pi_{sid,pid}((E \times \mathcal{P}) - (E \bowtie T))$$

Set join (**not only** case using natural join)

- To prove equality (1), we need to show that the following statements are logically equivalent

$$E(s, c) \wedge \neg T(c, p)$$

$$(E(s, c) \wedge p \in \mathcal{P}) \wedge \neg(E(s, c) \wedge T(c, p))$$

Since, clearly $p \in \mathcal{P}$ is true, we have to show that the following statements are equivalent:⁴

$$E(s, c) \wedge \neg T(c, p)$$

$$E(s, c) \wedge \neg(E(s, c) \wedge T(c, p))$$

⁴We are assuming that there is at least one professor.

Set join (**not only** case using natural join)

We need to show that the following statements are equivalent:

$$E(s, c) \wedge \neg T(c, p) \quad (\text{I})$$

$$E(s, c) \wedge \neg(E(s, c) \wedge T(c, p)) \quad (\text{II})$$

To see this, you need to determine if statements (I) and (II) have the same truth values. This is the case as shown in the following truth table:

$E(s, c)$	$T(c, p)$	Statement (I)	Statement (II)
true	true	false	false
true	false	true	true
false	true	false	false
false	false	false	false

Set join (**not all** case)

- Recall

$$\begin{aligned}\text{not all} &\leftrightarrow \{(s, p) | \exists c : c \in \mathcal{C}^{\bar{s}p}\} \\ &\leftrightarrow \{(s, p) | \exists c : c \in T(p) \wedge \neg(c \in E(s))\} \\ &\leftrightarrow \{(s, p) | \exists c : T(c, p) \wedge \neg E(s, c)\}\end{aligned}$$

- We first need to find an RA expression for the set

$$\{(s, c, p) | T(c, p) \wedge \neg E(s, c)\}.$$

$$\begin{aligned}\{(s, c, p) | T(c, p) \wedge \neg E(s, c)\} &= \{(s, c, p) | (s \in \mathcal{S} \wedge T(c, p)) \wedge \neg(E(s, c) \wedge T(c, p))\} \\ &= \{(s, c, p) | (s, c, p) \in (\mathcal{S} \times T) \wedge \neg((s, c, p) \in E \bowtie T)\} \\ &= (\mathcal{S} \times T) - (E \bowtie T)\end{aligned}\tag{1}$$

The equality (1) will be shown on the next slides.
Consequently, the RA expression for **not all** is

$$\pi_{sid,pid}((\mathcal{S} \times T) - (E \bowtie T))$$

Set join (**not all** case using natural join)

- To prove equality (1), we need to show that the following statements are logically equivalent

$$T(c, p)) \wedge \neg E(s, c)$$

$$(s \in S \wedge T(c, p)) \wedge \neg(E(s, c) \wedge T(c, p))$$

Since, clearly, $s \in S$ is true, we need to show that the following statements are equivalent:⁵

$$T(c, p) \wedge \neg E(s, c)$$

$$T(c, p) \wedge \neg(E(s, c) \wedge T(c, p))$$

⁵We assume that there is at least one student.

Set join (**not all** case using natural join)

We need to show that the following statements are equivalent:

$$T(c, p) \wedge \neg E(s, c) \quad (\text{I})$$

$$T(c, p) \wedge \neg(E(s, c) \wedge T(c, p)) \quad (\text{II})$$

To see this, you need to determine if statements (I) and (II) have the same truth values. This is the case as shown in the following truth table:

$E(s, c)$	$T(c, p)$	Statement (I)	Statement (II)
true	true	false	false
true	false	false	false
false	true	true	true
false	false	false	false

Set joins (not-empty cases)

- We have now developed the RA expressions for the following set joins:

$$\text{some} = \pi_{sid,pid}(E \bowtie T)$$

$$\text{not only} = \pi_{sid,pid}((E \times \mathcal{P}) - (E \bowtie T))$$

$$\text{not all} = \pi_{sid,pid}((S \times T) - (E \bowtie T))$$

- Notice that the time and space complexities for *not only* and *not all* are at least $O(|E| * |\mathcal{P}|)$ and $O(|T| * |\mathcal{S}|)$, respectively. (At IU, $160000 * 2000$ and $6000 * 40000$.)
- These set joins can be very expensive to evaluate.
- Observe that the time and space complexity for $E \bowtie S$ is never more than $O(|E| + |S| + |S| * |\mathcal{P}|)$ ⁶

⁶If we use the hash join algorithm.

Set joins (generalized intersection and difference)

A more compact way to remember these expressions is by using the following notations:

$$\begin{aligned} E \sqcap T &\leftrightarrow E \bowtie T \\ E \ominus T &\leftrightarrow (E \times \mathcal{P}) - (E \bowtie T) \\ T \ominus E &\leftrightarrow (\mathcal{S} \times T) - (E \bowtie T) \end{aligned}$$

With these notations, we get

$$\begin{aligned} \text{some} &= \pi_{sid,pid}(E \sqcap T) \\ \text{not only} &= \pi_{sid,pid}(E \ominus T) \\ \text{not all} &= \pi_{sid,pid}(T \ominus E) \end{aligned}$$

We call \sqcap the generalized intersection and \ominus the generalized difference.

Set joins (empty cases)

- Finding RA expressions for the empty cases is now easy: they consist of subtracting the corresponding not-empty cases from the set of all possible student-professor pairs (s, p) , i.e., the set $\mathcal{S} \times \mathcal{P}$.
- So, we get the following RA expressions for the set joins associated with **no**, **not only**, and **all**:

$$\begin{aligned}\text{no} &= \mathcal{S} \times \mathcal{P} - (\pi_{sid,pid}(E \sqcap T)) \\ \text{only} &= \mathcal{S} \times \mathcal{P} - (\pi_{sid,pid}(E \ominus T)) \\ \text{all} &= \mathcal{S} \times \mathcal{P} - (\pi_{sid,pid}(T \ominus E))\end{aligned}$$

Set joins (Summary)

The following are then the RA expressions for all of the cases considered:

some	=	$\pi_{sid,pid}(E \sqcap T)$
not only	=	$\pi_{sid,pid}(E \ominus T)$
not all	=	$\pi_{sid,pid}(T \ominus E)$
no	=	$\mathcal{S} \times \mathcal{P} - (\pi_{sid,pid}(E \sqcap T))$
only	=	$\mathcal{S} \times \mathcal{P} - (\pi_{sid,pid}(E \ominus T))$
all	=	$\mathcal{S} \times \mathcal{P} - (\pi_{sid,pid}(T \ominus E))$

Set joins **some** and **no** in SQL

- **some**

```
SELECT DISTINCT sid, pid  
FROM Enroll NATURAL JOIN TaughtBy
```

- **no**

```
SELECT sid, pid  
FROM Student CROSS JOIN Professor  
EXCEPT  
SELECT DISTINCT sid, pid  
FROM Enroll NATURAL JOIN TaughtBy
```

Set joins **not only** and **only** in SQL

not only

```
SELECT sid, pid
FROM   (SELECT sid, cno, pid
        FROM Enroll CROSS JOIN Professor
EXCEPT
        SELECT sid, cno, pid
        FROM Enroll NATURAL JOIN TaughtBy) q
```

only

```
SELECT sid, pid
FROM Student      CROSS JOIN Professor
EXCEPT
SELECT sid, pid
FROM   (SELECT sid, cno, pid
        FROM Enroll CROSS JOIN Professor
EXCEPT
        SELECT sid, cno, pid
        FROM Enroll NATURAL JOIN TaughtBy) q
```

Set joins **not all** and **all** in SQL

not all

```
SELECT sid, pid
FROM   (SELECT sid, cno, pid
        FROM Student CROSS JOIN TaughtBy
EXCEPT
        SELECT sid, cno, pid
        FROM Enroll NATURAL JOIN TaughtBy) q
```

all

```
SELECT sid, pid
FROM   Student      CROSS JOIN Professor
EXCEPT
SELECT sid, pid
FROM   (SELECT sid, cno, pid
        FROM Student CROSS JOIN TaughtBy
EXCEPT
        SELECT sid, cno, pid
        FROM Enroll NATURAL JOIN TaughtBy) q
```

The queries run in $O(|Student| * |TaughtBy| + |Enroll| * |TaughtBy|)$

Set joins **all** in Pure SQL

```
SELECT sid, pid
FROM Student
EXCEPT
SELECT sid, pid
FROM (SELECT sid, cno, pid
      FROM Student
      CROSS JOIN TaughtBy)
EXCEPT
SELECT sid, cno, pid
FROM Enroll
NATURAL JOIN TaughtBy q
```

Contrast this with the **all** set join expressed in Pure SQL

```
SELECT sid, pid
FROM Student s, Professor p
WHERE NOT EXISTS (SELECT 1
                   FROM TaughtBy t
                   WHERE t.pid = p.pid AND
                         t.cno NOT IN (SELECT e.cno
                                       FROM Enroll e
                                       WHERE e.sid = s.sid))
```

This query runs in $O(|Student| * |Professor| * |TaughtBy| * |Enroll|)$ and is thus orders of magnitude slower than the above RA SQL query which runs in $O(|Student| * |TaughtBy| + |Enroll| * |TaughtBy|)$.

Set joins (Example)

Consider the following queries:

"Find the student sid pairs (s_1, s_2) such that student s_1 takes [some|not only|not all|no|only|all] course(s) that student s_2 takes."

The queries are very similar to those we have been considering.

The only difference is that we now have to involve the *Enroll* relation twice: ones for student s_1 and the other for student s_2 .

Set joins (Example)

Let us denote by \tilde{E} the RA expression $\rho_{sid \rightarrow \widetilde{sid}}(\pi_{cno, sid}(E))$. So \tilde{E} is the same as E , except that its schema is (cno, \widetilde{sid}) instead of (sid, cno) . We have renamed sid by \widetilde{sid} .

For this example, the **search space** is the set of all student pairs $\mathcal{S} \times \mathcal{S}$ and we assume its schema is $(\widetilde{sid}, \widetilde{sid})$.

Then the RA expressions for the above six queries are

some	=	$\pi_{sid, \widetilde{sid}}(E \bowtie \tilde{E})$
not only	=	$\pi_{sid, \widetilde{sid}}(E \ominus \tilde{E})$
not all	=	$\pi_{sid, \widetilde{sid}}(\tilde{E} \ominus E)$
no	=	$\mathcal{S} \times \mathcal{S} - (\pi_{sid, \widetilde{sid}}(E \bowtie \tilde{E}))$
only	=	$\mathcal{S} \times \mathcal{S} - (\pi_{sid, \widetilde{sid}}(E \ominus \tilde{E}))$
all	=	$\mathcal{S} \times \mathcal{S} - (\pi_{sid, \widetilde{sid}}(\tilde{E} \ominus E))$

More set joins

- The RA region expressions $E \sqcap T$, $E \ominus T$, and $T \ominus E$ permit us to express many more set joins. We will consider one such example next.
- "Find the student-professor pairs (s, p) such that student s takes at least 2 courses not taught by professor p ."
- More formally, we need to find a RA expression for the query

$$\{(s, p) \mid \exists c_1, c_2 : c_1 \neq c_2 \wedge c_1 \in (E(s) - T(p)) \wedge c_2 \in (E(s) - T(p))\}$$

- Alternatively,

$$\{(s, p) \mid \exists c_1, c_2 : c_1 \neq c_2 \wedge \{c_1, c_2\} \subseteq (E(s) - T(p))\}$$

More set joins

- "Find the student-professor pairs (s, p) such that student s takes at least 2 courses not taught by professor p ."

$$\{(s, p) \mid \exists c_1, c_2 : c_1 \neq c_2 \wedge \{c_1, c_2\} \subseteq (E(s) - T(p))\}$$

- From this formulation, it is clear that we need to involve the region expression $E \ominus T$, but we need two copies of it, one to reason about c_1 and the other to reason about c_2 .
- Let these two copies be $E_1 \ominus T_1$ with schema (sid, cno_1, pid) and $E_2 \ominus T_2$ with schema (sid, cno_2, pid) .
- Then the RA expression for the query is as follows:

$$\pi_{sid,pid}(\sigma_{cno_1 \neq cno_2}((E_1 \ominus T_1) \bowtie (E_2 \ominus T_2)))$$

Set semijoins (Motivation)

- Consider the relations $\text{Enroll}(\text{sid}, \text{cno})$ and the relation of CS courses $\text{CS}(\text{cno})$.
- We are interested in students satisfying some property.
- "Find each student sid s such that student s takes [*some|not only|not all|no|only|all*] CS course(s)."

Set semijoins (Solution)

An analysis very similar to that for set joins gives the following RA queries for these queries.

First observe that $E \bowtie CS = E \ltimes CS$.

some	=	$\pi_{sid}(E \bowtie CS)$	=	$\pi_{sid}(E \ltimes CS)$
not only	=	$\pi_{sid}(E \ominus CS)$		
not all	=	$\pi_{sid}(CS \ominus E)$		
no	=	$S - (\pi_{sid}(E \ltimes CS))$		
only	=	$S - (\pi_{sid}(E \ominus CS))$		
all	=	$S - (\pi_{sid}(CS \ominus E))$		

Where $E \ominus CS = E - (E \ltimes CS) = E \overline{\ltimes} CS$

$$CS \ominus E = (S \times CS) - (E \ltimes CS)$$

Set semijoins (Efficiency consideration)

Reconsider the 6 semijoins:

$$\begin{aligned}\text{some} &= \pi_{sid}(E \ltimes CS) \\ \text{not only} &= \pi_{sid}(E \overline{\ltimes} CS)) \\ \text{not all} &= \pi_{sid}((\pi_{sid}(S) \times CS) - (E \ltimes CS)) \\ \text{no} &= \pi_{sid}(S) - (\pi_{sid}(E \ltimes CS)) \\ \text{only} &= \pi_{sid}(S) - \pi_{sid}(E \overline{\ltimes} CS) \\ \text{all} &= \pi_{sid}(S) - \pi_{sid}((\pi_{sid}(S) \times CS) - (E \ltimes CS))\end{aligned}$$

Let $k = |S|$, $l = |CS|$, and $n = |E|$.

- Then, using hashing methods, the **some**, **no**, **not only**, **only** set semijoins can be implemented in **linear time and space** $O(|k| + |l| + |n|)$. (I.e., extremely fast.)
- However, the **not all** and **all** set semijoins require **quadratic time and space** $O((k * l) + n)$ because of the need to compute $\pi_{sid}(S) \times CS$. (I.e., can be extremely expensive.)

Set semijoin **some** and **no** in SQL

- **some**

```
SELECT DISTINCT sid
FROM Enroll NATURAL JOIN CS
```

- **no**

```
SELECT sid
FROM Student
EXCEPT
SELECT DISTINCT sid
FROM Enroll NATURAL JOIN CS
```

Set semijoin **not only** in SQL

- **not only**

```
SELECT DISTINCT sid
  FROM (SELECT sid, cno
        FROM Enroll
      EXCEPT
      SELECT sid, cno
        FROM Enroll NATURAL JOIN CS) q
```

Set semijoin **only** SQL

- **only**

```
SELECT sid
FROM Student
EXCEPT
SELECT DISTINCT sid
FROM (SELECT sid, cno
      FROM Enroll
EXCEPT
      SELECT sid, cno
      FROM Enroll NATURAL JOIN CS) q
```

Set semijoin **not all** in SQL

- **not all**

```
SELECT DISTINCT sid
  FROM (SELECT sid, cno
        FROM (SELECT sid from Student) s CROSS JOIN CS
      EXCEPT
      SELECT sid, cno
        FROM Enroll NATURAL JOIN CS) q
```

Set semijoin **all** in SQL

- **all**

```
SELECT    sid
FROM      Student s
EXCEPT
SELECT    DISTINCT sid
FROM      (SELECT sid, cno
          FROM  (SELECT sid from Student) s CROSS JOIN CS
EXCEPT
SELECT    sid, cno
FROM      Enroll NATURAL JOIN CS) q
```

Set joins (General case)

- Let $E_1(A_1, \dots, A_m, C_1, \dots, C_k)$ and $E_2(C_1, \dots, C_k, B_1, \dots, B_n)$ be RA expressions.
- We assume that each attribute has an associated domain: these are denoted $\mathcal{A}_1, \dots, \mathcal{A}_m, \mathcal{C}_1, \dots, \mathcal{C}_k, \mathcal{B}_1, \dots, \mathcal{B}_n$.
- We can then build RA expressions $E_1 \sqcap E_2$, $E_1 \ominus E_2$, and $E_2 \ominus E_1$ as follows:

$$E_1 \sqcap E_2 = E_1 \bowtie E_2$$

$$E_1 \ominus E_2 = (E_1 \times \mathcal{B}_1 \times \dots \times \mathcal{B}_n) - (E_1 \bowtie E_2)$$

$$E_2 \ominus E_1 = (\mathcal{A}_1 \times \dots \times \mathcal{A}_m \times E_2) - (E_1 \bowtie E_2)$$

- Using these region expressions we can build numerous other set-joins.

Set joins (From general case to special cases)

$$E_1 \sqcap E_2 = E_1 \bowtie E_2$$

$$E_1 \ominus E_2 = (E_1 \times \mathcal{B}_1 \times \cdots \times \mathcal{B}_n) - (E_1 \bowtie E_2)$$

$$E_2 \ominus E_1 = (\mathcal{A}_1 \times \cdots \times \mathcal{A}_m \times E_2) - (E_1 \bowtie E_2)$$

We can now consider special cases:

- When $m = k = n = 1$ we have the region expressions for the set joins we have seen above.
- When $m = k = 1$ and $n = 0$ we have the region expressions for semi set joins we have seen above.
- When $m = n = 0$, then

$$E_1 \sqcap E_2 = E_1 \cap E_2$$

$$E_1 \ominus E_2 = E_1 - E_2$$

$$E_2 \ominus E_1 = E_2 - E_1$$

Translating SQL Queries to Relational Algebra Expressions

Dirk Van Gucht¹

¹Indiana University

February 26, 2019

Outline

- **Objective:** Discuss an algorithm that translates a SQL query into an equivalent RA expression
- **Motivation:** Translate a declaratively specified query into a procedurally specified query
- **Restriction:** We do not attempt to get an efficient RA expression. Finding an equivalent efficient RA expression is done during **query optimization**

Strategy

- SQL queries with set predicates will be translated to equivalent SQL queries without set predicates
- WHERE conditions will be eliminated by translating them into FROM clauses using selections and join operations, or by decomposing them into more basic components (then translate these) and then use the set operations union, intersection, and set difference
- These SQL queries will then be translated into RA expressions

Query Forms (Basic case)

```
SELECT [DISTINCT] L( $t_1, \dots, t_n$ )
FROM   R1  $t_1, \dots, R_n t_n$ 
WHERE  C( $t_1, \dots, t_n$ )
```

- $L(t_1, \dots, t_n)$ is a list of (named) components of the tuple variables t_1 through t_n ¹
- R_1 through R_n are either relations or non-parameterized SQL queries aliased by the tuple variables t_1 through t_n
- $C(t_1, \dots, t_n)$ is any valid SQL condition involving the components of the variables t_1 through t_n
- We do not handle SQL queries with aggregate functions or queries with subqueries in the FROM clause

¹There may also appear constants in L ; these need special treatment

Query Forms (queries with set operations)

Assuming that Q , Q_1 , and Q_2 are SQL queries, we consider the following SQL query forms

$$\begin{array}{c} Q_1 \\ \text{UNION} \\ Q_2 \end{array}$$
$$\begin{array}{c} Q_1 \\ \text{INTERSECT} \\ Q_2 \end{array}$$
$$\begin{array}{c} Q_1 \\ \text{EXCEPT} \\ Q_2 \end{array}$$
$$(Q)$$

Discussion: interaction between projection π and set operations \cup , \cap , and $-$

- Before we can start with the translation algorithm, it is crucial to discuss how the projection operator π interacts with the set operations \cup , \cap , and $-$.
- Understanding this is vital for the correct translations of SQL **WHERE** clauses that use the **OR**, **AND**, and **NOT** boolean operations
- We will consider the following interactions:
 - Projection π and union \cup
 - Projection π and intersection \cap
 - Projection π and set difference $-$

Projection π distributes over \cup

Given RA expressions E_1 and E_2 , it is the case that

$$\pi_L(E_1 \cup E_2) = \pi_L(E_1) \cup \pi_L(E_2)$$

An application of this is the following: (C_1 and C_2 are some conditions)

$$\pi_L(\sigma_{C_1 \vee C_2}(E)) = \pi_L(\sigma_{C_1}(E) \cup \sigma_{C_2}(E)) = \pi_L(\sigma_{C_1}(E)) \cup \pi_L(\sigma_{C_2}(E))$$

Translating OR in WHERE clause

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE C1(t1, ..., tn) OR C2(t1, ..., tn)
```

can be translated to become

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE C1(t1, ..., tn)
UNION
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE C2(t1, ..., tn)
```

Projection π does not distribute over \cap

Given RA expressions E_1 and E_2 , it is the case that

$$\pi_L(E_1 \cap E_2) \subseteq \pi_L(E_1) \cap \pi_L(E_2)$$

But there exist cases where

$$\pi_L(E_1 \cap E_2) \neq \pi_L(E_1) \cap \pi_L(E_2)$$

This complexity features in reasoning about the expression $\pi_L(\sigma_{C_1 \wedge C_2}(E))$ since

$$\pi_L(\sigma_{C_1 \wedge C_2}(E)) = \pi_L(\sigma_{C_1}(E) \cap \sigma_{C_2}(E)) \subseteq \pi_L(\sigma_{C_1}(E)) \cap \pi_L(\sigma_{C_2}(E))$$

But there are cases where

$$\pi_L(\sigma_{C_1 \wedge C_2}(E)) \neq \pi_L(\sigma_{C_1}(E)) \cap \pi_L(\sigma_{C_2}(E))$$

Projection π does **not** distribute over intersection \cap

R		
a	b	c
1	1	2
1	2	1

$$\pi_a(\sigma_{b=1 \wedge c=1}(R))$$

a

correct

$$\pi_a(\sigma_{b=1}(R) \cap \sigma_{c=1}(R))$$

a

incorrect

$$\pi_a(\sigma_{b=1}(R)) \cap \pi_a(\sigma_{c=1}(R))$$

a
1

Projection π does not distribute over intersection \cap (Predicate Logic)

The underlying reason why projection π does not distribute over intersection \cap is that an existential quantifier \exists does not distribute over a conjunction \wedge in Predicate Logic

$$\begin{aligned}\pi_a(\sigma_{b=1 \wedge c=1}(R)) \\ &= \\ \pi_a(\sigma_{b=1}(R) \cap \sigma_{c=1}(R)) \\ &= \\ \{a \mid \exists b \exists c [(R(a, b, c) \wedge b = 1) \wedge (R(a, b, c) \wedge c = 1)]\} \\ &\quad \textcolor{red}{\subset} \\ \{a \mid \exists b \exists c (R(a, b, c) \wedge b = 1) \wedge \exists b \exists c (R(a, b, c) \wedge c = 1)\} \\ &= \\ \pi_{b=1}(R) \cap \pi_{c=1}(R)\end{aligned}$$

Translating AND in WHERE clause (Correct)

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE C1(t1, ..., tn) AND C2(t1, ..., tn)
```

can be translated to become

```
SELECT Lq(t1, ..., tn)2
FROM (SELECT t1.* , ..., tn.*
       FROM R1 t1, ..., Rn tn
       WHERE C1(t1, ..., tn)
       INTERSECT
       SELECT t1.* , ..., tn.*
       FROM R1 t1, ..., Rn tn
       WHERE C2(t1, ..., tn)
) q
```

² $L^q(t_1, \dots, t_n)$ indicates that the components of t_1 through t_n in L may need to be renamed as components of q

Translating AND in WHERE clause (Incorrect)

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE C1(t1, ..., tn) AND C2(t1, ..., tn)
```

is not equivalent³ with

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE C1(t1, ..., tn)
INTERSECT
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE C2(t1, ..., tn)
```

³Projection does not distribute over intersection

Translating AND in WHERE clause (Correct)

```
SELECT e.sid  
FROM Enroll e  
WHERE e.cno = 100 AND e.grade = 'A'
```

can be translated to become

```
SELECT q.sid  
FROM (SELECT e.sid, e.cno, e.grade  
      FROM Enroll e  
      WHERE e.cno = 100  
INTERSECT  
      SELECT e.sid, e.cno, e.grade  
      FROM Enroll e  
      WHERE e.grade = 'A'  
) q
```

Translating AND in WHERE clause (Incorrect)

```
SELECT e.sid  
FROM Enroll e  
WHERE e.cno = 100 AND e.grade = 'A'
```

is not equivalent⁴ with

```
SELECT e.sid  
FROM Enroll e  
WHERE e.cno = 100  
INTERSECT  
SELECT e.sid  
FROM Enroll e  
WHERE e.grade = 'A'
```

⁴Projection does not distribute over intersection

Projection π does not distribute over $-$

Given RA expressions E_1 and E_2 , it is the case that

$$\pi_L(E_1 - E_2) \supseteq \pi_L(E_1) - \pi_L(E_2)$$

But there exist cases where

$$\pi_L(E_1 - E_2) \neq \pi_L(E_1) - \pi_L(E_2)$$

This complexity features in reasoning about the expression $\pi_L(\sigma_{\neg C}(E))$ since

$$\pi_L(\sigma_{\neg C}(E)) = \pi_L(E - \sigma_C(E)) \supseteq \pi_L(E) - \pi_L(\sigma_C(E))$$

But there are cases where

$$\pi_L(\sigma_{\neg C}(E)) \neq \pi_L(E) - \pi_L(\sigma_C(E))$$

Projection π does **not** distribute over set difference –

R	
a	b
1	1
1	2

$$\pi_a(\sigma_{\neg(b=1)}(R))$$

a
1

correct

$$\pi_a(R - \sigma_{b=1}(R))$$

a
1

incorrect

$$\pi_a(R) - \pi_a(\sigma_{b=1}(R))$$

a

Projection π does not distribute over set difference – (Predicate Logic)

The underlying reason why projection π does not distribute over distribution \cap is that an existential quantifier \exists does not distribute over a conjunction-negation $\wedge \neg$ sequence in Predicate Logic

$$\begin{aligned}\pi_a(\sigma_{\neg(b=1)}(R)) \\ &= \\ \pi_a(R - \sigma_{b=1}(R)) \\ &= \\ \{a \mid \exists b (R(a, b) \wedge \neg(R(a, b) \wedge b = 1))\} \\ &\supsetneq \\ \{a \mid \exists b R(a, b) \wedge \neg \exists b (R(a, b) \wedge b = 1)\} \\ &= \\ \pi_a(R) - \pi_a(\sigma_{b=1}(R))\end{aligned}$$

Translating NOT in WHERE clause (Correct)

```
SELECT L(t1, ..., tn)
FROM   R1 t1, ..., Rn tn
WHERE  NOT C(t1, ..., tn)
```

can be translated to become

```
SELECT Lq(t1, ..., tn)
FROM   (SELECT   t1.* , ..., tn.*
        FROM     R1 t1, ..., Rn tn
        EXCEPT
        SELECT   t1.* , ..., tn.*
        FROM     R1 t1, ..., Rn tn
        WHERE    C(t1, ..., tn)
      ) q
```

Translating NOT in WHERE clause (Incorrect)

```
SELECT  L(t1, ..., tn)
FROM    R1 t1, ..., Rn tn
WHERE   NOT C(t1, ..., tn)
```

is not equivalent⁵ with

```
SELECT  L(t1, ..., tn)
FROM    R1 t1, ..., Rn tn
EXCEPT
SELECT  L(t1, ..., tn)
FROM    R1 t1, ..., Rn tn
WHERE   C(t1, ..., tn)
```

⁵Projection do not distribute over set difference

Translating NOT in WHERE clause (Correct)

```
SELECT e.sid  
FROM Enroll e  
WHERE NOT e.grade = 'A'
```

can be translated to become

```
SELECT q.sid  
FROM (SELECT e.sid, e.cno, e.grade  
      FROM Enroll e  
      EXCEPT  
      SELECT e.sid, e.cno, e.grade  
      FROM Enroll e  
      WHERE e.grade = 'A'  
    ) q
```

Translating NOT in WHERE clause (Incorrect)

```
SELECT e.sid  
FROM Enroll e  
WHERE NOT e.grade = 'A'
```

is not equivalent⁶ with

```
SELECT e.sid  
FROM Enroll e  
EXCEPT  
SELECT e.sid  
FROM Enroll e  
WHERE e.grade = 'A'
```

⁶Projection does not distribute over set difference

Projection π does not distribute over set difference –

R		
A	B	C
1	1	2
1	2	1

$$\pi_A(\sigma_{B=1 \wedge \neg(C=1)}(R))$$

A
1

correct

$$\pi_A(\sigma_{B=1}(R) - \sigma_{C=1}(R))$$

A
1

incorrect

$$\pi_A(\sigma_{B=1}(R)) - \pi_A(\sigma_{C=1}(R))$$

A

Translating AND NOT in WHERE clause (Correct)

```
SELECT  L( $t_1, \dots, t_n$ )
FROM     $R_1 t_1, \dots, R_n t_n$ 
WHERE    $C_1(t_1, \dots, t_n)$  AND NOT  $C_2(t_1, \dots, t_n)$ 
```

can be translated to become

```
SELECT   $L^q(t_1, \dots, t_n)$ 
FROM    (SELECT       $t_1.*, \dots, t_n.*$ 
          FROM         $R_1 t_1, \dots, R_n t_n$ 
          WHERE        $C_1(t_1, \dots, t_n)$ 
          EXCEPT
          SELECT       $t_1.*, \dots, t_n.*$ 
          FROM         $R_1 t_1, \dots, R_n t_n$ 
          WHERE        $C_2(t_1, \dots, t_n)$ 
        ) q
```

Translating AND NOT in WHERE clause (Incorrect)

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE C1(t1, ..., tn) AND NOT C2(t1, ..., tn)
```

is not equivalent⁷ with

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE C1(t1, ..., tn)
EXCEPT
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE C2(t1, ..., tn)
```

⁷Projection do not distribute over set difference

Translating AND NOT in WHERE clause (Correct)

```
SELECT e.sid  
FROM Enroll e  
WHERE e.cno = 100 AND NOT e.grade = 'A'
```

can be translated to become

```
SELECT q.sid  
FROM (SELECT e.sid, e.cno, e.grade  
      FROM Enroll e  
      WHERE e.cno = 100  
      EXCEPT  
      SELECT e.sid, e.cno, e.grade  
      FROM Enroll e  
      WHERE e.grade = 'A'  
) q
```

Translating AND NOT in WHERE clause (Incorrect)

```
SELECT e.sid  
FROM Enroll e  
WHERE e.cno = 100 AND NOT e.grade = 'A'
```

is not equivalent⁸ with

```
SELECT e.sid  
FROM Enroll e  
WHERE e.cno = 100  
EXCEPT  
SELECT e.sid  
FROM Enroll e  
WHERE e.grade = 'A'
```

⁸Projection does not distribute over set difference

Translating AND NOT in WHERE clause (Also correct)

```
SELECT  L( $t_1, \dots, t_n$ )
FROM     $R_1 t_1, \dots, R_n t_n$ 
WHERE    $C_1(t_1, \dots, t_n)$  AND NOT  $C_2(t_1, \dots, t_n)$ 
```

can be also be translated to become

```
SELECT  Lq( $t_1, \dots, t_n$ )
FROM    (SELECT       $t_1.*, \dots, t_n.*$ 
        FROM        $R_1 t_1, \dots, R_n t_n$ 
        WHERE      $C_1(t_1, \dots, t_n)$ 
        EXCEPT
        SELECT       $t_1.*, \dots, t_n.*$ 
        FROM        $R_1 t_1, \dots, R_n t_n$ 
        WHERE      $C_1(t_1, \dots, t_n)$  AND  $C_2(t_1, \dots, t_n)$ 
) q
```

Translating AND NOT in WHERE clause (Also correct)

```
SELECT e.sid  
FROM Enroll e  
WHERE e.cno = 100 AND NOT e.grade = 'A'
```

can be translated to become

```
SELECT q.sid  
FROM (SELECT e.sid, e.cno, e.grade  
      FROM Enroll e  
      WHERE e.cno = 100  
      EXCEPT  
      SELECT e.sid, e.cno, e.grade  
      FROM Enroll e  
      WHERE e.cno= 100 AND e.grade = 'A'  
      ) q
```

Translating EXISTS in WHERE clause (Example)

Let $R(a, b)$ and $S(b, c)$ be two relations.

```
SELECT r.a
FROM R r
WHERE EXISTS (SELECT S.c
               FROM S s
               WHERE r.b = s.b)
```

In Predicate Logic,

$$\begin{aligned} & \{a \mid \exists b (R(a, b) \wedge \{c \mid S(b, c)\} \neq \emptyset)\} \\ &= \\ & \{a \mid \exists b (R(a, b) \wedge \exists c S(b, c))\} \\ &= \\ & \{a \mid \exists b \exists c (R(a, b) \wedge S(b, c))\} \\ &= \\ & \pi_{R.a}(\sigma_{R.b=S.b}(R \times S)) \end{aligned}$$

Translating EXISTS in WHERE clause (Example in Predicate Logic)

$$\begin{aligned} & \{a \mid \exists b (R(a, b) \wedge \{c \mid S(b, c)\} \neq \emptyset)\} \\ &= \\ & \{a \mid \exists b (R(a, b) \wedge \exists c S(b, c))\} \\ &= \\ & \{a \mid \exists b \exists c (R(a, b) \wedge S(b, c))\} \\ &= \\ & \pi_{R.a}(\sigma_{R.b=S.b}(R \times S)) \end{aligned}$$

In SQL,

```
SELECT DISTINCT r.a
FROM R r, S s
WHERE r.b = s.b
```

Translating EXISTS in WHERE clause (Example)

Let $R(a, c)$ and $S(b, c)$ be two relations.

```
SELECT r.a
FROM R r
WHERE EXISTS (SELECT S.c
               FROM S s
               WHERE r.b = s.b)
```

is translated to

```
SELECT DISTINCT r.a
FROM R r, S s
WHERE r.b = s.b
```

Translating EXISTS in WHERE clause (General case)

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE EXISTS (SELECT 1
               FROM S1 u1, ..., Sm um
               WHERE C(u1, ..., um, t1, ..., tn))
```

is translated to

```
SELECT DISTINCT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn, S1 u1, ..., Sm um
WHERE C(u1, ..., um, t1, ..., tn)
```

Translating EXISTS in WHERE clause

```
SELECT s.sid
FROM Student s
WHERE EXISTS (SELECT 1
               FROM Enroll e, Course c
               WHERE e.sid = s.sid AND e.cno = c.cno AND c.dept = 'CS')
```

is translated to

```
SELECT DISTINCT s.sid
FROM Student s, Enroll e, Course c
WHERE e.sid = s.sid AND e.cno = c.cno AND c.dept = 'CS'
```

Translating NOT EXISTS in WHERE clause (Example)

Let $R(a, b)$ and $S(b, c)$ be two relations.

```
SELECT r.a  
FROM R r  
WHERE NOT EXISTS (SELECT S.c  
                   FROM S s  
                   WHERE r.b = s.b)
```

In Predicate Logic,

$$\begin{aligned} & \{a \mid \exists b (R(a, b) \wedge \{c \mid S(b, c)\} = \emptyset)\} \\ &= \\ & \{a \mid \exists b (R(a, b) \wedge \neg \exists c S(b, c))\} \\ &= \\ & \{a \mid \exists b (R(a, b) \wedge \neg \exists c (R(a, b) \wedge S(b, c)))\} \\ &= \\ & \pi_{R.a}(R - \pi_{R.a, R.b}(\sigma_{R.b=S.b}(R \times S))) \end{aligned}$$

Translating NOT EXISTS in WHERE clause (Example)

$$\begin{aligned} & \{a \mid \exists b (R(a, b) \wedge \{c \mid S(b, c)\} = \emptyset)\} \\ &= \\ & \pi_{R.a}(R - \pi_{R.a, R.b}(\sigma_{R.b=S.b}(R \times S))) \end{aligned}$$

In SQL,

```
SELECT DISTINCT q.a
FROM   (SELECT      R.a, R.b
        FROM        R r
        EXCEPT
        SELECT      R.a, R.b
        FROM        R r, S s
        WHERE       r.b = s.b
      ) q
```

Translating NOT EXISTS in WHERE clause (Example)

```
SELECT r.a
FROM R r
WHERE NOT EXISTS (SELECT S.c
                   FROM S s
                   WHERE r.b = s.b)
```

is translated to

```
SELECT DISTINCT q.a
FROM (SELECT      R.a, R.b
      FROM       R r
      EXCEPT
      SELECT      R.a, S.b
      FROM       R r, S s
      WHERE      r.b = s.b
) q
```

Translating NOT EXISTS in WHERE clause (General case)

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE NOT EXISTS (SELECT 1
                   FROM S1 u1, ..., Sm um
                   WHERE C(u1, ..., um, t1, ..., tn))
```

is translated to

```
SELECT DISTINCT Lq(t1, ..., tn)
FROM (SELECT t1.* , ..., tn.*
       FROM R1 t1, ..., Rn tn
       EXCEPT
       SELECT t1.* , ..., tn.*
       FROM R1 t1, ..., Rn tn, S1 u1, ..., Sm um
       WHERE C(u1, ..., um, t1, ..., tn)
     ) q
```

Translating NOT EXISTS in WHERE clause

```
SELECT s.sid  
FROM Student s  
WHERE NOT EXISTS (SELECT 1  
                   FROM Enroll e  
                   WHERE e.sid = s.sid AND e.grade = 'A')
```

is translated to

```
SELECT q.sid  
FROM (SELECT s.sid, s.sname  
      FROM Student s  
      EXCEPT  
      SELECT s.sid, s.sname  
      FROM Student s, Enroll e  
      WHERE e.sid = s.sid AND e.grade = 'A'  
) q
```

Translating AND NOT EXISTS in WHERE clause

```
SELECT s.sid  
FROM Student s  
WHERE s.sname = 'Ann' AND  
      NOT EXISTS (SELECT 1  
                  FROM Enroll e  
                  WHERE e.sid = s.sid AND e.grade = 'A')
```

is translated to

```
SELECT q.sid  
FROM (SELECT s.sid, s.sname  
      FROM Student s  
      WHERE s.sname = 'Ann'  
      EXCEPT  
      SELECT s.sid, s.sname  
      FROM Student s, Enroll e  
      WHERE e.sid = s.sid AND e.grade = 'A'  
    ) q
```

Translating AND NOT EXISTS in WHERE clause (Alternative)

```
SELECT s.sid
FROM Student s
WHERE s.sname = 'Ann' AND
NOT EXISTS (SELECT 1
            FROM Enroll e
            WHERE e.sid = s.sid AND e.grade = 'A')
```

is translated to

```
SELECT q.sid
FROM (SELECT s.sid, s.sname
      FROM Student s
      WHERE s.sname = 'Ann'
EXCEPT
      SELECT s.sid, s.sname
      FROM Student s, Enroll e
      WHERE s.sname='Ann' AND e.sid = s.sid AND e.grade = 'A'
) q
```

Translating IN in WHERE clause

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE (ti1.Aj1, ..., tik.Ajk) IN
      (SELECT ul1.Bm1, ..., ulk.Bmk
       FROM S1 u1, ..., Sm um
       WHERE C(u1, ..., um, t1, ..., tn))
```

is translated to

```
SELECT DISTINCT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn, S1 u1, ..., Sm um
WHERE C(u1, ..., um, t1, ..., tn) AND
      ti1.Aj1 = ul1.Bm1 AND ... AND tik.Ajk = ulk.Bmk
```

Translating NOT IN in WHERE clause

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE (ti1.Aj1, ..., tik.Ajk) NOT IN
      (SELECT ul1.Bm1, ..., ulk.Bmk
       FROM S1 u1, ..., Sm um
       WHERE C(u1, ..., um, t1, ..., tn))
```

is translated to

```
SELECT Lq(t1, ..., tn)
FROM (SELECT t1.* , ..., tn.*
       FROM R1 t1, ..., Rn tn
       EXCEPT
       SELECT t1.* , ..., tn.*
       FROM R1 t1, ..., Rn tn, S1 u1, ..., Sm um
       WHERE C(u1, ..., um, t1, ..., tn) AND
             ti1.Aj1 = ul1.Bm1 AND ... AND tik.Ajk = ulk.Bmk
) q
```

Translating θ SOME in WHERE clause

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE ti1.Aj1  $\theta$  SOME
      (SELECT ul1.Bm1
       FROM S1 u1, ..., Sm um
       WHERE C(u1, ..., um, t1, ..., tn))
```

is translated to

```
SELECT DISTINCT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn, S1 u1, ..., Sm um
WHERE C(u1, ..., um, t1, ..., tn) AND
      ti1.Aj1  $\theta$  ul1.Bm1
```

Translating θ ALL in WHERE clause

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE ti1.Aj1  $\theta$  ALL
      (SELECT ui1.Bm1
       FROM S1 u1, ..., Sm um
       WHERE C(u1, ..., um, t1, ..., tn))
```

is translated to

```
SELECT Lq(t1, ..., tn)
FROM (SELECT t1.* , ..., tn.*
       FROM R1 t1, ..., Rn tn
       EXCEPT
       SELECT t1.* , ..., tn.*
       FROM R1 t1, ..., Rn tn, S1 u1, ..., Sm um
       WHERE C(u1, ..., um, t1, ..., tn) AND
             NOT ti1.Aj1  $\theta$  ui1.Bm1
      ) q
```

Translating θ ALL in WHERE clause

```
SELECT p.pid
FROM Person p
WHERE p.age ≤ ALL
      (SELECT p1.age
       FROM Person p1)
```

is translated to

```
SELECT DISTINCT q.pid
FROM (SELECT p.pid, p.age
      FROM Person p
EXCEPT
      SELECT p.pid, p.age
      FROM Person p, Person p1
      WHERE NOT p.age ≤ p1.age
) q
```

Translating SQL with doubly-nested set predicates

“Find the sid of each student who is only enrolled CS courses.”

Eliminate first-level NOT EXISTS predicate

“Find the sid of each student who is only enrolled in CS courses.”

Eliminate second-level AND NOT EXISTS

"Find the sid of each student who is only enrolled in CS courses."

```
SELECT      q1.sid
FROM        (SELECT s.sid AS sid, s.sname
            FROM  Student s
            EXCEPT
            SELECT q2.sid, s.sname
            FROM (SELECT s.sid AS sid, s.sname, e.sid, e.cno, e.grade
                  FROM  Student s, Enroll e
                  WHERE e.sid = s.sid
                  EXCEPT
                  (SELECT s.sid, s.sname, e.sid, e.cno, e.grade
                  FROM  Student s, Enroll e, Course c
                  WHERE e.cno = c.cno AND c.dept = 'CS' ) q2) q1
```

Translating SQL with doubly-nested set predicates

“Find the sid of each student who is enrolled in all CS courses.”

Eliminate second-level AND NOT EXISTS

"Find the sid of each student who is enrolled in all CS courses."

```
SELECT q1.ssid
FROM (SELECT s.sid AS ssid, s.sname
      FROM Student s
      EXCEPT
      SELECT q2.ssid, s.sname
      FROM (SELECT s.sid AS ssid, s.sname, c.cno, c.dname
            FROM Student s, Course c
            WHERE c.dname = 'CS'
            EXCEPT
            (SELECT s.sid, s.sname, c.cno, c.dname
             FROM Student s, Enroll e, Course c
             WHERE e.cno = c.cno AND e.sid = s.sid ) q2) q1
```

Moving **WHERE** condition to **FROM** clause

- In the previous slides we have shown how set predicates can be translated
- After that process, we will have SQL queries wherein the **WHERE** clauses consist of boolean combinations of conditions of the form
 - $t.A \theta a$; or
 - $t_i.A \theta t_j.B$
- In the following slides we will show how these **WHERE** clauses can be moved to **FROM** clauses
- We might also have queries without a **WHERE** clause and/or without a **FROM** clause; these require special treatment

SQL queries without **WHERE** and **FROM** clauses

SELECT a AS A

This query is translated to the RA expression

$(A : a)$

SQL queries without **FROM** clauses

```
SELECT L(t1, ..., tn)
FROM   R1 t1, ..., Rn tn
```

This query is translated to the RA expression

$$\pi_{L(t_1, \dots, t_n)}(R_1 \times \dots \times R_n)$$

Moving WHERE condition to FROM clause (condition on at least three relations)

Assume that the condition C applies to at least three relations, i.e., the condition is of the form $C(t_{i_1}, t_{i_2}, t_{i_3}, \dots, t_{i_k})$ with $k \geq 3$

```
SELECT L(t1, ..., tn)
FROM R1 t1, R2 t2, R3 t3, ..., Rn tn
WHERE C(ti1, ti2, ti3, ..., tik)
```

We can now introduce the CROSS JOIN in the FROM clause by replacing each ‘,’ with CROSS JOIN

```
SELECT L(t1, ..., tn)
FROM R1 t1 CROSS JOIN R2 t2 CROSS JOIN R3 t3 CROSS JOIN ... CROSS JOIN Rn tn
WHERE C(ti1, ti2, ti3, ..., tik)
```

In RA,

$$\pi_{L(t_1, \dots, t_n)}(\sigma_{C(t_{i_1}, t_{i_2}, t_{i_3}, \dots, t_{i_k})}(R_1 \times R_2 \times R_3 \times \dots \times R_n))$$

Moving WHERE condition to FROM clause (condition on at least three relations) Example

```
SELECT L(t1, t2, t3)
FROM R1 t1, R2 t2, R3 t3
WHERE t1.A1 θ1 t2.A2 OR t2.A3 θ2 t3.A4
```

is translated to

$$\pi_{L(t_1, t_2, t_3)}(\sigma_{t_1.A_1 \theta_1 t_2.A_2 \vee t_2.A_3 \theta_2 t_3.A_4}(R_1 \times R_2 \times R_3))$$

Moving WHERE condition to FROM clause (condition on single relation)

```
SELECT L(t1, ..., tn)
FROM R1 t1, R2 t2, ..., Ri ti, ..., Rn tn
WHERE C(ti) [AND C'(ti1, ..., tik)]
```

Observe that $C(t_i)$ is only a condition on R_i . This query is translated to

```
SELECT L(t1, ..., tn)
FROM R1 t1, R2 t2 ...,
      (SELECT ti.* FROM Ri WHERE C(ti)) ti, ..., Rn tn
[WHERE C'(ti1, ..., tik)]
```

Moving WHERE condition to FROM clause (condition on single relation)

```
SELECT L(t1, ..., tn)
FROM   R1 t1, R2 t2, ...,
       (SELECT ti.* FROM Ri WHERE C(ti)) ti, ..., Rn tn
[WHERE  C'(ti1, ..., tik)]
```

We can now introduce the CROSS JOIN in the FROM clause by replacing each ‘,’ with CROSS JOIN

```
SELECT L(t1, ..., tn)
FROM   R1 t1 CROSS JOIN R2 t2 CROSS JOIN ... CROSS JOIN
       (SELECT ti.* FROM Ri WHERE C(ti)) ti CROSS JOIN ... CROSS JOIN Rn tn
[WHERE  C'(ti1, ..., tik)]
```

Which, in the notation of RA, corresponds to the expression

$$\pi_{L(t_1, \dots, t_n)}(\sigma_{C'(t_{i_1}, \dots, t_{i_k})}(R_1 \times R_2 \times \dots \times \sigma_{C(t_i)}(R_i) \times \dots \times R_n))$$

or, when $C'(t_{i_1}, \dots, t_{i_k})$ is missing,

$$\pi_{L(t_1, \dots, t_n)}(R_1 \times R_2 \times \dots \times \sigma_{C(t_i)}(R_i) \times \dots \times R_n)$$

Moving WHERE condition to FROM clause (condition on two relations)

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Ri ti, ..., Rj tj, ..., Rn tn
WHERE C(ti, tj) [AND C'(ti1, ..., tik)]
```

Observe that $C(t_i, t_j)$ is a condition relating R_i and R_j . This query is translated to

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Ri-1 tj-1, Ri+1 ti+1, ..., Rj-1 tj-1, Rj+1 tj+1, ..., Rn tn,
      Ri ti JOIN Rj tj ON C(ti, tj)
[WHERE C'(ti1, ..., tik)]
```

Moving WHERE condition to FROM clause (condition on two relations)

```
SELECT L(t1, ..., tn)
FROM   R1 t1, ..., Ri-1 tj-1, Ri+1 ti+1, ..., Rj-1 tj-1, Rj+1 tj+1, ... Rn tn,
       Ri ti JOIN Rj tj ON C(ti, tj)
[WHERE C'(ti1, ..., tik)]
```

Recalling that each ‘,’ in the FROM clause corresponds to a CROSS JOIN, this query can be formulated in RA as follows:

$$\pi_{L(t_1, \dots, t_n)}(\sigma_{C'(t_{i_1}, \dots, t_{i_k})}(R_1 \times \dots \times R_{i-1} \times R_{i+1} \times \dots \times R_{j-1} \times R_{j+1} \times \dots \times R_n \times (R_i \bowtie_{C(t_i, t_j)} R_j)))$$

or, when $C'(t_{i_1}, \dots, t_{i_k})$ is missing

$$\pi_{L(t_1, \dots, t_n)}(R_1 \times \dots \times R_{i-1} \times R_{i+1} \times \dots \times R_{j-1} \times R_{j+1} \times \dots \times R_n \times (R_i \bowtie_{C(t_i, t_j)} R_j))$$

Moving WHERE condition to FROM clause (natural join condition on two relations)

Assume that A_1, \dots, A_k are the common attributes of R_i and R_j and that $C(t_i, t_j)$ is the condition

$$t_i.A_1 = t_j.A_1 \text{ AND } \dots \text{ AND } t_i.A_k = t_j.A_k$$

then

```
SELECT L(t1, ..., tn)
FROM   R1 t1, ..., Ri-1 tj-1, Ri+1 ti+1, ..., Rj-1 tj-1, Rj+1 tj+1, ..., Rn tn,
       Rj ti JOIN Rj tj ON C(ti, tj)
[WHERE C'(ti1, ..., tik)]
```

is translated to

```
SELECT L(t1, ..., tn)
FROM   R1 t1, ..., Ri-1 tj-1, Ri+1 ti+1, ..., Rj-1 tj-1, Rj+1 tj+1, ..., Rn tn,
       Rj ti NATURAL JOIN Rj tj
[WHERE C'(ti1, ..., tik)]
```

In RA,

$$\pi_{L(t_1, \dots, t_n)}(\sigma_{C'(t_{i_1}, \dots, t_{i_k})}(R_1 \times \dots \times R_{i-1} \times R_{i+1} \times \dots \times R_{j-1} \times R_{j+1} \times \dots \times R_n \times (R_i \bowtie R_j)))$$

SQL queries with set operations **UNION**, **INTERSECT**, or **EXCEPT**

Assuming Q_1 and Q_2 SQL queries, the queries of the form

$$\begin{matrix} Q_1 \\ \text{UNION [INTERSECT | EXCEPT]} \\ Q_2 \end{matrix}$$

can be translated to RA as follows

$$E_{Q_1} \cup [\cap | -] E_{Q_2}$$

where E_{Q_1} and E_{Q_2} are the RA expressions corresponding to Q_1 and Q_2

Example

“Find the sid of each student who is only enrolled in CS courses.”

```
SELECT    q1.sid
FROM      (SELECT s.sid AS sid, s.sname
           FROM Student s
         EXCEPT
           SELECT q2.sid, s.sname
             FROM (SELECT s.sid AS sid, s.sname, e.sid, e.cno
                   FROM Student s, Enroll e
                  WHERE e.sid = s.sid
                EXCEPT
                  (SELECT s.sid, s.sname, e.sid, e.cno
                    FROM Student s, Enroll e, Course c
                   WHERE e.cno = c.cno AND c.dept = 'CS' ) q2) q1
```

Example

“Find the sid of each student who is only enrolled in CS courses.”

```
SELECT      q1.sid
FROM        (SELECT s.sid AS ssid, s.sname
            FROM Student s
           EXCEPT
           SELECT q2.sid, s.sname
            FROM (SELECT s.sid AS ssid, s.sname, e.sid, e.cno
                  FROM Student s NATURAL JOIN Enroll e
                 EXCEPT
                 (SELECT s.sid, s.sname, e.sid, e.cno
                  FROM Student s CROSS JOIN Enroll E
                  NATURAL JOIN (SELECT c.* FROM Course c WHERE dept = 'CS') c) q2) q1
```

is translated to

$$\pi_{sid}(\pi_{sid, sname}(S) - \pi_{S.sid, sname}(\pi_{S.sid, sname, E.sid, cno}(S \bowtie E) - \pi_{S.sid, sname, E.sid, cno}(S \times E \bowtie \sigma_{dept='CS'}(C))))$$

where S , E , and C denote Student, Enroll, and Course, respectively.

Example (Optimization)

“Find the sid of each student who is only enrolled in CS courses.”

$$\pi_{sid}(\pi_{sid, sname}(S) - \pi_{S.sid, sname}(\pi_{S.sid, sname, E.sid, cno}(S \bowtie E) - \pi_{S.sid, sname, E.sid, cno}(S \times E \bowtie \sigma_{dept='CS'}(C))))$$

This can be **optimized** to the RA expression

$$\pi_{sid}(S) - \pi_{sid}(\pi_{sid, cno}(E) - \pi_{sid, cno}(E) \ltimes \pi_{cno}(\sigma_{dept='CS'}(C)))$$

Notice that this is the RA expression for the **only** set semijoin
If furthermore the schema of Enroll is (sid,cno), this expression becomes

$$\pi_{sid}(S) - \pi_{sid}(E - E \ltimes CS) = \pi_{sid} - \pi_{sid}(E \overline{\ltimes} CS)$$

where *CS* denotes the RA expression $\pi_{cno}(\sigma_{dept='CS'}(C))$

Translating SQL Queries to Relational Algebra Expressions

Dirk Van Gucht¹

¹Indiana University

February 26, 2019

Outline

- **Objective:** Discuss an algorithm that translates a SQL query into an equivalent RA expression
- **Motivation:** Translate a declaratively specified query into a procedurally specified query
- **Restriction:** We do not attempt to get an efficient RA expression. Finding an equivalent efficient RA expression is done during **query optimization**

Strategy

- SQL queries with set predicates will be translated to equivalent SQL queries without set predicates
- WHERE conditions will be eliminated by translating them into FROM clauses using selections and join operations, or by decomposing them into more basic components (then translate these) and then use the set operations union, intersection, and set difference
- These SQL queries will then be translated into RA expressions

Query Forms (Basic case)

```
SELECT [DISTINCT] L( $t_1, \dots, t_n$ )
FROM   R1  $t_1, \dots, R_n t_n$ 
WHERE  C( $t_1, \dots, t_n$ )
```

- $L(t_1, \dots, t_n)$ is a list of (named) components of the tuple variables t_1 through t_n ¹
- R_1 through R_n are either relations or **non-parameterized** SQL queries aliased by the tuple variables t_1 through t_n
- $C(t_1, \dots, t_n)$ is any valid SQL condition involving the components of the variables t_1 through t_n
- We do not handle SQL queries with aggregate functions or queries with subqueries in the FROM clause

¹There may also appear constants in L ; these need special treatment

Query Forms (queries with set operations)

Assuming that Q , Q_1 , and Q_2 are SQL queries, we consider the following SQL query forms

$$\begin{array}{c} Q_1 \\ \text{UNION} \\ Q_2 \end{array}$$
$$\begin{array}{c} Q_1 \\ \text{INTERSECT} \\ Q_2 \end{array}$$
$$\begin{array}{c} Q_1 \\ \text{EXCEPT} \\ Q_2 \end{array}$$

Discussion: interaction between projection π and set operations \cup , \cap , and $-$

- Before we can start with the translation algorithm, it is crucial to discuss how the projection operator π interacts with the set operations \cup , \cap , and $-$.
- Understanding this is vital for the correct translations of SQL **WHERE** clauses that use the **OR**, **AND**, and **NOT** boolean operations
- We will consider the following interactions:
 - Projection π and union \cup
 - Projection π and intersection \cap
 - Projection π and set difference $-$

Projection π distributes over \cup

Given RA expressions E_1 and E_2 , it is the case that

$$\pi_L(E_1 \cup E_2) = \pi_L(E_1) \cup \pi_L(E_2)$$

An application of this is the following: (C_1 and C_2 are some conditions)

$$\pi_L(\sigma_{C_1 \vee C_2}(E)) = \pi_L(\sigma_{C_1}(E) \cup \sigma_{C_2}(E)) = \pi_L(\sigma_{C_1}(E)) \cup \pi_L(\sigma_{C_2}(E))$$

Translating **OR** in **WHERE** clause

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE C1(t1, ..., tn) OR C2(t1, ..., tn)
```

can be translated to become

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE C1(t1, ..., tn)
UNION
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE C2(t1, ..., tn)
```

Projection π does not distribute over \cap

Given RA expressions E_1 and E_2 , it is the case that

$$\pi_L(E_1 \cap E_2) \subseteq \pi_L(E_1) \cap \pi_L(E_2)$$

But there exist cases where,

$$\pi_L(E_1 \cap E_2) \neq \pi_L(E_1) \cap \pi_L(E_2)$$

This complexity features in reasoning about the expression $\pi_L(\sigma_{C_1 \wedge C_2}(E))$ since

$$\pi_L(\sigma_{C_1 \wedge C_2}(E)) = \pi_L(\sigma_{C_1}(E) \cap \sigma_{C_2}(E)) \subseteq \pi_L(\sigma_{C_1}(E)) \cap \pi_L(\sigma_{C_2}(E))$$

But there are cases where,

$$\pi_L(\sigma_{C_1 \wedge C_2}(E)) \neq \pi_L(\sigma_{C_1}(E)) \cap \pi_L(\sigma_{C_2}(E))$$

Projection π does **not** distribute over intersection \cap

R		
a	b	c
1	1	2
1	2	1

$$\pi_a(\sigma_{b=1 \wedge c=1}(R))$$

a

correct

$$\pi a(\sigma_{b=1}(R) \cap \sigma_{c=1}(R))$$

a

incorrect

$$\pi a(\sigma_{b=1}(R)) \cap \pi a(\sigma_{c=1}(R))$$

a
1

Translating AND in WHERE clause (Correct)

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE C1(t1, ..., tn) AND C2(t1, ..., tn)
```

can be translated to become

```
SELECT Lq(t1, ..., tn)2
FROM (SELECT t1.* ..., tn.*
       FROM R1 t1, ..., Rn tn
       WHERE C1(t1, ..., tn)
     INTERSECT
     SELECT t1.* ..., tn.*
       FROM R1 t1, ..., Rn tn
       WHERE C2(t1, ..., tn)
   ) q
```

²L^q(t₁, ..., t_n) indicates that the components of t₁ through t_n in L may need to be renamed as components of q

Translating AND in WHERE clause (Incorrect)

```
SELECT      L(t1, ..., tn)
FROM        R1 t1, ..., Rn tn
WHERE       C1(t1, ..., tn) AND C2(t1, ..., tn)
```

is not equivalent³ with

```
SELECT      L(t1, ..., tn)
FROM        R1 t1, ..., Rn tn
WHERE       C1(t1, ..., tn)
INTERSECT
SELECT      L(t1, ..., tn)
FROM        R1 t1, ..., Rn tn
WHERE       C2(t1, ..., tn)
```

³Projection does not distribute over intersection

Translating AND in WHERE clause (Correct)

```
SELECT e.sid  
FROM Enroll e  
WHERE e.cno = 100 AND e.grade = 'A'
```

can be translated to become

```
SELECT q.sid  
FROM (SELECT e.sid, e.cno, e.grade  
      FROM Enroll e  
      WHERE e.cno = 100  
INTERSECT  
      SELECT e.sid, e.cno, e.grade  
      FROM Enroll e  
      WHERE e.grade = 'A'  
) q
```

Translating AND in WHERE clause (Incorrect)

```
SELECT e.sid  
FROM Enroll e  
WHERE e.cno = 100 AND e.grade = 'A'
```

is not equivalent⁴ with

```
SELECT e.sid  
FROM Enroll e  
WHERE e.cno = 100  
INTERSECT  
SELECT e.sid  
FROM Enroll e  
WHERE e.grade = 'A'
```

⁴Projection does not distribute over intersection

Projection π does not distribute over $-$

Given RA expressions E_1 and E_2 , it is the case that

$$\pi_L(E_1 - E_2) \supseteq \pi_L(E_1) - \pi_L(E_2)$$

But there exist cases where

$$\pi_L(E_1 - E_2) \neq \pi_L(E_1) - \pi_L(E_2)$$

This complexity features in reasoning about the expression $\pi_L(\sigma_{\neg C}(E))$ since

$$\pi_L(\sigma_{\neg C}(E)) = \pi_L(E - \sigma_C(E)) \supseteq \pi_L(E) - \pi_L(\sigma_C(E))$$

But there are cases where

$$\pi_L(\sigma_{\neg C}(E)) \neq \pi_L(E) - \pi_L(\sigma_C(E))$$

Projection π does **not** distribute over set difference –

R	
a	b
1	1
1	2

$$\pi_a(\sigma_{\neg(b=1)}(R))$$

a
1

correct

$$\pi_a(R - \sigma_{b=1}(R))$$

a
1

incorrect

$$\pi_a(R) - \pi_a(\sigma_{b=1}(R))$$

a

Translating NOT in WHERE clause (Correct)

```
SELECT L(t1, ..., tn)
FROM   R1 t1, ..., Rn tn
WHERE  NOT C(t1, ..., tn)
```

can be translated to become

```
SELECT Lq(t1, ..., tn)
FROM   (SELECT   t1.* ..., tn.*
        FROM     R1 t1, ..., Rn tn
        EXCEPT
        SELECT   t1.* ..., tn.*
        FROM     R1 t1, ..., Rn tn
        WHERE   C(t1, ..., tn)
      ) q
```

Translating NOT in WHERE clause (Incorrect)

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE NOT C(t1, ..., tn)
```

is not equivalent⁵ with

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
EXCEPT
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE C(t1, ..., tn)
```

⁵Projection do not distribute over set difference

Translating NOT in WHERE clause (Correct)

```
SELECT e.sid  
FROM Enroll e  
WHERE NOT e.grade = 'A'
```

can be translated to become

```
SELECT q.sid  
FROM (SELECT e.sid, e.cno, e.grade  
      FROM Enroll e  
      EXCEPT  
      SELECT e.sid, e.cno, e.grade  
      FROM Enroll e  
      WHERE e.grade = 'A'  
    ) q
```

Translating NOT in WHERE clause (Incorrect)

```
SELECT e.sid  
FROM Enroll e  
WHERE NOT e.grade = 'A'
```

is not equivalent⁶ with

```
SELECT e.sid  
FROM Enroll e  
EXCEPT  
SELECT e.sid  
FROM Enroll e  
WHERE e.grade = 'A'
```

⁶Projection does not distribute over set difference

Projection π does not distribute over set difference –

R		
A	B	C
1	1	2
1	2	1

$$\pi_A(\sigma_{B=1 \wedge \neg(C=1)}(R))$$

A
1

correct

$$\pi_A(\sigma_{B=1}(R) - \sigma_{C=1}(R))$$

A
1

incorrect

$$\pi_A(\sigma_{B=1}(R)) - \pi_A(\sigma_{C=1}(R))$$

A

Translating AND NOT in WHERE clause (Correct)

```
SELECT L( $t_1, \dots, t_n$ )
FROM   R1  $t_1, \dots, R_n t_n$ 
WHERE  C1( $t_1, \dots, t_n$ ) AND NOT C2( $t_1, \dots, t_n$ )
```

can be translated to become

```
SELECT Lq( $t_1, \dots, t_n$ )
FROM   (SELECT       $t_1.* \dots, t_n.*$ 
        FROM
        WHERE   C1( $t_1, \dots, t_n$ )
        EXCEPT
        SELECT       $t_1.* \dots, t_n.*$ 
        FROM
        WHERE   C2( $t_1, \dots, t_n$ )
      ) q
```

Translating AND NOT in WHERE clause (Incorrect)

```
SELECT L(t1, ..., tn)
FROM   R1 t1, ..., Rn tn
WHERE  C1(t1, ..., tn) AND NOT C2(t1, ..., tn)
```

is not equivalent⁷ with

```
SELECT L(t1, ..., tn)
FROM   R1 t1, ..., Rn tn
WHERE  C1(t1, ..., tn)
EXCEPT
SELECT L(t1, ..., tn)
FROM   R1 t1, ..., Rn tn
WHERE  C2(t1, ..., tn)
```

⁷Projection do not distribute over set difference

Translating AND NOT in WHERE clause (Correct)

```
SELECT e.sid  
FROM Enroll e  
WHERE e.cno = 100 AND NOT e.grade = 'A'
```

can be translated to become

```
SELECT q.sid  
FROM (SELECT e.sid, e.cno, e.grade  
      FROM Enroll e  
      WHERE e.cno = 100  
      EXCEPT  
      SELECT e.sid, e.cno, e.grade  
      FROM Enroll e  
      WHERE e.grade = 'A'  
) q
```

Translating AND NOT in WHERE clause (Incorrect)

```
SELECT e.sid  
FROM Enroll e  
WHERE e.cno = 100 AND NOT e.grade = 'A'
```

is not equivalent⁸ with

```
SELECT e.sid  
FROM Enroll e  
WHERE e.cno = 100  
EXCEPT  
SELECT e.sid  
FROM Enroll e  
WHERE e.grade = 'A'
```

⁸Projection does not distribute over set difference

Translating AND NOT in WHERE clause (Also correct)

```
SELECT L(t1, ..., tn)
FROM   R1 t1, ..., Rn tn
WHERE  C1(t1, ..., tn) AND NOT C2(t1, ..., tn)
```

can be also be translated to become

```
SELECT Lq(t1, ..., tn)
FROM   (SELECT t1.* ..., tn.*
        FROM   R1 t1, ..., Rn tn
        WHERE  C1(t1, ..., tn)
        EXCEPT
        SELECT t1.* ..., tn.*
        FROM   R1 t1, ..., Rn tn
        WHERE  C1(t1, ..., tn) AND C2(t1, ..., tn)
      ) q
```

Translating AND NOT in WHERE clause (Also correct)

```
SELECT e.sid  
FROM Enroll e  
WHERE e.cno = 100 AND NOT e.grade = 'A'
```

can be translated to become

```
SELECT q.sid  
FROM (SELECT e.sid, e.cno, e.grade  
      FROM Enroll e  
      WHERE e.cno = 100  
      EXCEPT  
      SELECT e.sid, e.cno, e.grade  
      FROM Enroll e  
      WHERE e.cno= 100 AND e.grade = 'A'  
) q
```

Translating EXISTS in WHERE clause (Example)

Let $R(a, b)$ and $S(b, c)$ be two relations.

```
SELECT r.a
FROM R r
WHERE EXISTS (SELECT S.c
               FROM S s
               WHERE r.b = s.b)
```

In Predicate Logic,

$$\begin{aligned} & \{a \mid \exists b (R(a, b) \wedge \{c \mid S(b, c)\} = \emptyset)\} \\ &= \\ & \{a \mid \exists b (R(a, b) \wedge \exists c S(b, c))\} \\ &= \\ & \{a \mid \exists b \exists c (R(a, b) \wedge S(b, c))\} \\ &= \\ & \pi_{R.a}(\sigma_{R.b=S.b}(R \times S)) \end{aligned}$$

Translating EXISTS in WHERE clause (Example in Predicate Logic)

$$\begin{aligned}\{a \mid \exists b (R(a, b) \wedge \{c \mid S(b, c)\} = \emptyset)\} \\= \\ \{a \mid \exists b (R(a, b) \wedge \exists c S(b, c))\} \\= \\ \{a \mid \exists b \exists c (R(a, b) \wedge S(b, c))\} \\= \\ \pi_{R.a}(\sigma_{R.b=S.b}(R \times S))\end{aligned}$$

In SQL,

```
SELECT DISTINCT r.a
FROM   R r, S s
WHERE  r.b = s.b
```

Translating EXISTS in WHERE clause (Example)

Let $R(a, b)$ and $S(b, c)$ be two relations.

```
SELECT r.a
FROM R r
WHERE EXISTS (SELECT S.c
               FROM S s
               WHERE r.b = s.b)
```

is translated to

```
SELECT DISTINCT r.a
FROM R r, S s
WHERE r.b = s.b
```

Translating EXISTS in WHERE clause (General case)

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE EXISTS (SELECT 1
               FROM S1 u1, ..., Sm um
               WHERE C(u1, ..., um, t1, ..., tn))
```

is translated to

```
SELECT DISTINCT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn, S1 u1, ..., Sm um
WHERE C(u1, ..., um, t1, ..., tn)
```

Translating **EXISTS** in **WHERE** clause

```
SELECT    s.sid
FROM      Student s
WHERE     EXISTS (SELECT 1
                  FROM    Enroll e, Course c
                  WHERE   e.sid = s.sid AND e.cno = c.cno AND c.dept = 'CS')
```

is translated to

```
SELECT    DISTINCT s.sid
FROM      Student s, Enroll e, Course c
WHERE     e.sid = s.sid AND e.cno = c.cno AND c.dept = 'CS'
```

Translating NOT EXISTS in WHERE clause (Example)

Let $R(a, b)$ and $S(b, c)$ be two relations.

```
SELECT  r.a
FROM    R r
WHERE   NOT EXISTS (SELECT S.c
                      FROM   S s
                      WHERE  r.b = s.b)
```

In Predicate Logic,

$$\{a \mid \exists b(R(a, b) \wedge \{c \mid S(b, c)\} = \emptyset)\}$$
$$=$$

$$\{a \mid \exists b(R(a, b) \wedge \neg \exists c S(b, c)))\}$$
$$=$$

$$\{a \mid \exists b(R(a, b) \wedge \neg \exists c (R(a, b) \wedge S(b, c)))\}$$
$$=$$

$$\pi_{R.a}(R - \pi_{R.a, R.b}(\sigma_{R.b = S.b}(R \times S)))$$

Translating NOT EXISTS in WHERE clause (Example)

$$\begin{aligned} & \{a \mid \exists b(R(a, b) \wedge \{c \mid S(b, c)\} = \emptyset)\} \\ &= \\ & \pi_{R.a}(R - \pi_{R.a, R.b}(\sigma_{R.b = S.b}(R \times S))) \end{aligned}$$

In SQL,

```
SELECT DISTINCT q.a
FROM   (SELECT      R.a, R.b
        FROM        R r
        EXCEPT
        SELECT      R.a, R.b
        FROM        R r, S s
        WHERE       r.b = s.b
      ) q
```

Translating NOT EXISTS in WHERE clause (Example)

```
SELECT r.a
FROM R r
WHERE NOT EXISTS (SELECT S.c
                   FROM S s
                   WHERE r.b = s.b)
```

is translated to

```
SELECT DISTINCT q.a
FROM (SELECT R.a, R.b
      FROM R r
      EXCEPT
      SELECT R.a, S.b
      FROM R r, S s
      WHERE r.b = s.b
    ) q
```

Translating NOT EXISTS in WHERE clause (General case)

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE NOT EXISTS (SELECT 1
                   FROM S1 u1, ..., Sm um
                   WHERE C(u1, ..., um, t1, ..., tn))
```

is translated to

```
SELECT DISTINCT Lq(t1, ..., tn)
FROM (SELECT t1.* ..., tn.*
       FROM R1 t1, ..., Rn tn
       EXCEPT
       SELECT t1.* ..., tn.*
       FROM R1 t1, ..., Rn tn, S1 u1, ..., Sm um
       WHERE C(u1, ..., um, t1, ..., tn)
     ) q
```

Translating NOT EXISTS in WHERE clause

```
SELECT s.sid  
FROM Student s  
WHERE NOT EXISTS (SELECT 1  
                   FROM Enroll e  
                   WHERE e.sid = s.sid AND e.grade = 'A')
```

is translated to

```
SELECT q.sid  
FROM (SELECT s.sid, s.sname  
      FROM Student s  
      EXCEPT  
      SELECT s.sid, s.sname  
      FROM Student s, Enroll e  
      WHERE e.sid = s.sid AND e.grade = 'A'  
) q
```

Translating AND NOT EXISTS in WHERE clause

```
SELECT s.sid  
FROM Student s  
WHERE s.sname = 'Ann' AND  
      NOT EXISTS (SELECT 1  
                  FROM Enroll e  
                  WHERE e.sid = s.sid AND e.grade = 'A')
```

is translated to

```
SELECT q.sid  
FROM (SELECT s.sid, s.sname  
      FROM Student s  
      WHERE s.sname = 'Ann'  
      EXCEPT  
      SELECT s.sid, s.sname  
      FROM Student s, Enroll e  
      WHERE e.sid = s.sid AND e.grade = 'A'  
) q
```

Translating AND NOT EXISTS in WHERE clause (Alternative)

```
SELECT    s.sid
FROM      Student s
WHERE     s.sname = 'Ann' AND
          NOT EXISTS (SELECT 1
                        FROM   Enroll e
                        WHERE e.sid = s.sid AND e.grade = 'A')
```

is translated to

```
SELECT    q.sid
FROM      (SELECT s.sid, s.sname
           FROM Student s
           WHERE s.sname = 'Ann'
EXCEPT
           SELECT s.sid, s.sname
           FROM Student s, Enroll e
           WHERE s.sname='Ann' AND e.sid = s.sid AND e.grade = 'A'
) q
```

Translating IN in WHERE clause

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE (ti1.Aj1, ..., tik.Ajk) IN
      (SELECT ul1.Bm1, ..., ulk.Bmk
       FROM S1 u1, ..., Sm um
       WHERE C(u1, ..., um, t1, ..., tn))
```

is translated to

```
SELECT DISTINCT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn, S1 u1, ..., Sm um
WHERE C(u1, ..., um, t1, ..., tn) AND
      ti1.Aj1 = ul1.Bm1 AND ... AND tik.Ajk = ulk.Bmk
```

Translating NOT IN in WHERE clause

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE (t1.Aj1, ..., tk.Ajk) NOT IN
      (SELECT u1.Bm1, ..., ulk.Bmk
       FROM S1 u1, ..., Sm um
       WHERE C(u1, ..., um, t1, ..., tn))
```

is translated to

```
SELECT Lq(t1, ..., tn)
FROM (SELECT t1.* ..., tn.*
       FROM R1 t1, ..., Rn tn
       EXCEPT
       SELECT t1.* ..., tn.*
       FROM R1 t1, ..., Rn tn, S1 u1, ..., Sm um
       WHERE C(u1, ..., um, t1, ..., tn) AND
             t1.Aj1 = u1.Bm1 AND ... AND tk.Ajk = uk.Bmk
) q
```

Translating θ SOME in WHERE clause

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE ti1.Aj1  $\theta$  SOME
      (SELECT ul1.Bm1
       FROM S1 u1, ..., Sm um
       WHERE C(u1, ..., um, t1, ..., tn))
```

is translated to

```
SELECT DISTINCT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn, S1 u1, ..., Sm um
WHERE C(u1, ..., um, t1, ..., tn) AND
      ti1.Aj1  $\theta$  ul1.Bm1
```

Translating θ ALL in WHERE clause

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE ti1.Aj1  $\theta$  ALL
      (SELECT ul1.Bm1
       FROM S1 u1, ..., Sm um
       WHERE C(u1, ..., um, t1, ..., tn))
```

is translated to

```
SELECT Lq(t1, ..., tn)
FROM (SELECT t1.* ..., tn.*
      FROM R1 t1, ..., Rn tn
      EXCEPT
      SELECT t1.* ..., tn.*
      FROM R1 t1, ..., Rn tn, S1 u1, ..., Sm um
      WHERE C(u1, ..., um, t1, ..., tn) AND
            NOT ti1.Aj1  $\theta$  ul1.Bm1
      ) q
```

Translating θ ALL in WHERE clause

```
SELECT p.pid
FROM Person p
WHERE p.age ≤ ALL
      (SELECT p1.age
       FROM Person p1)
```

is translated to

```
SELECT DISTINCT q.pid
FROM (SELECT p.pid, p.age
      FROM Person p
      EXCEPT
      SELECT p.pid, p.age
      FROM Person p, Person p1
      WHERE NOT p.age ≤ p1.age
) q
```

Moving **WHERE** condition to **FROM** clause

- In the previous slides we have shown how set predicates can be translated
- After that process, we will have SQL queries wherein the **WHERE** clauses consist of boolean combinations of conditions of the form
 - $t.A \theta a$; or
 - $t_i.A \theta t_j.B$
- In the following slides we will show how these **WHERE** clauses can be moved to **FROM** clauses
- We might also have queries without a **WHERE** clause and/or without a **FROM** clause; these require special treatment

SQL queries without **WHERE** and **FROM** clauses

SELECT a AS A

This query is translated to the RA expression

$(A : a)$

SQL queries without **WHERE** clause

```
SELECT L(t1, ..., tn)
FROM   R1 t1, ..., Rn tn
```

This query is translated to the RA expression

$$\pi_{L(t_1, \dots, t_n)}(R_1 \times \dots \times R_n)$$

Moving WHERE condition to FROM clause (condition on at least three relations)

Assume that the condition C applies to at least three relations, i.e., the condition is of the form $C(t_{i_1}, t_{i_2}, t_{i_3}, \dots, t_{i_k})$ with $k \geq 3$

```
SELECT L(t1, ..., tn)
FROM R1 t1, R2 t2, R3 t3, ..., Rn tn
WHERE C(ti1, ti2, ti3, ..., tik)
```

We can now introduce the CROSS JOIN in the FROM clause by replacing each ‘,’ with CROSS JOIN

```
SELECT L(t1, ..., tn)
FROM R1 t1 CROSS JOIN R2 t2 CROSS JOIN R3 t3 CROSS JOIN ... CROSS JOIN Rn tn
WHERE C(ti1, ti2, ti3, ..., tik)
```

In RA,

$$\pi_{L(t_1, \dots, t_n)}(\sigma_{C(t_{i_1}, t_{i_2}, t_{i_3}, \dots, t_{i_k})}(R_1 \times R_2 \times R_3 \times \dots \times R_n))$$

Moving WHERE condition to FROM clause (condition on at least three relations) Example

```
SELECT L(t1, t2, t3)
FROM R1 t1, R2 t2, R3 t3
WHERE t1.A1 θ1 t2.A2 OR t2.A3 θ2 t3.A4
```

is translated to

$$\pi_{L(t_1, t_2, t_3)}(\sigma_{t_1.A_1 \theta_1 t_2.A_2 \text{ OR } t_2.A_3 \theta_2 t_3.A_4}(R_1 \times R_2 \times R_3))$$

Moving WHERE condition to FROM clause (condition on single relation)

```
SELECT L(t1, ..., tn)
FROM R1 t1, R2 t2, ..., Ri ti, ..., Rn tn
WHERE C(ti) [AND Cl(ti1, ..., tik)]
```

Observe that $C(t_i)$ is only a condition on R_i . This query is translated to

```
SELECT L(t1, ..., tn)
FROM R1 t1, R2 t2...
      (SELECT ti.* FROM Ri WHERE C(ti)) ti, ..., Rn tn
[WHERE Cl(ti1, ..., tik)]
```

Moving WHERE condition to FROM clause (condition on single relation)

```
SELECT L(t1, ..., tn)
FROM   R1 t1, R2 t2, ...
        (SELECT ti.* FROM Ri WHERE C(ti)) ti, ..., Rn tn
[WHERE Ct(t1, ..., tk)]
```

We can now introduce the CROSS JOIN in the FROM clause by replacing each ',' with CROSS JOIN

```
SELECT L(t1, ..., tn)
FROM   R1 t1 CROSS JOIN R2 t2 CROSS JOIN ... CROSS JOIN
        (SELECT ti.* FROM Ri WHERE C(ti)) ti CROSS JOIN ... CROSS JOIN Rn tn
[WHERE Ct(t1, ..., tk)]
```

Which, in the notation of RA, corresponds to the expression

$$\pi_{L(t_1, \dots, t_n)}(\sigma_{C^t(t_1, \dots, t_k)}(R_1 \times R_2 \times \dots \times \sigma_{C(t_i)}(R_i) \times \dots \times R_n))$$

or, when $C^l(t_1, \dots, t_k)$ is missing,

$$\pi_{L(t_1, \dots, t_n)}(R_1 \times R_2 \times \dots \times \sigma_{C(t_i)}(R_i) \times \dots \times R_n)$$

Moving WHERE condition to FROM clause (condition on two relations)

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Ri ti, ..., Rj tj, ..., Rn tn
WHERE C(ti, tj) [AND Cl(ti1, ..., tik)]
```

Observe that $C(t_i, t_j)$ is a condition relating R_i and R_j . This query is translated to

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Ri-1 tj-1, Ri+1 ti+1, ..., Rj-1 tj-1, Rj+1 tj+1, ... Rn tn,
      Ri ti JOIN Rj tj ON C(ti, tj)
[WHERE Ct(ti1, ..., tik)]
```

Moving WHERE condition to FROM clause (condition on two relations)

```
SELECT L(t1, ..., tn)
FROM   R1 t1, ..., Ri-1 tj-1, Ri+1 ti+1, ..., Rj-1 tj-1, Rj+1 tj+1, ..., Rn tn,
       Ri ti JOIN Rj tj ON C(ti, tj)
[WHERE Ct(ti1, ..., tik)]
```

Recalling that each ‘,’ in the FROM clause corresponds to a CROSS JOIN, this query can be formulated in RA as follows:

$$\pi L(t_1, \dots, t_n)(\sigma_{C^t(t_{i_1}, \dots, t_{i_k})}(R_1 \times \dots \times R_{i-1} \times R_{i+1} \times \dots \times R_{j-1} \times R_{j+1} \times \dots \times R_n \times (R_i \times C(t_i, t_j) \times R_j)))$$

or, when $C^l(t_{i_1}, \dots, t_{i_k})$ is missing

$$\pi L(t_1, \dots, t_n)(R_1 \times \dots \times R_{i-1} \times R_{i+1} \times \dots \times R_{j-1} \times R_{j+1} \times \dots \times R_n \times (R_i \times C(t_i, t_j) \times R_j))$$

Moving WHERE condition to FROM clause (natural join condition on two relations)

Assume that A_1, \dots, A_k are the common attributes of R_i and R_j and that $C(t_i, t_j)$ is the condition

$$t_i.A_1 = t_j.A_1 \text{ AND } \dots \text{ AND } t_i.A_k = t_j.A_k$$

then

```
SELECT L(t1, ..., tn)
FROM   R1 t1, ..., Ri-1 tj-1, Ri+1 ti+1, ..., Rj-1 tj-1, Rj+1 tj+1, ..., Rn tn,
       Ri ti JOIN Rj tj ON C(ti, tj)
[WHERE Ct(t1, ..., tk)]
```

is translated to

```
SELECT L(t1, ..., tn)
FROM   R1 t1, ..., Ri-1 tj-1, Ri+1 ti+1, ..., Rj-1 tj-1, Rj+1 tj+1, ..., Rn tn,
       Ri ti NATURAL JOIN Rj tj
[WHERE Ct(t1, ..., tk)]
```

In RA,

$$\pi L(t_1, \dots, t_n) (\sigma_{C^t(t_{j_1}, \dots, t_{j_k})} (R_1 \times \dots \times R_{i-1} \times R_{i+1} \times \dots \times R_{j-1} \times R_{j+1} \times \dots \times R_n \times (R_i \times R_j)))$$

SQL queries with set operations **UNION**, **INTERSECT**, or **EXCEPT**

Assuming Q_1 and Q_2 SQL queries, the queries of the form

$$\begin{matrix} Q_1 \\ \text{UNION [INTERSECT | EXCEPT]} \\ Q_2 \end{matrix}$$

can be translated to RA as follows

$$E_{Q_1} \textcolor{red}{U} [\textcolor{red}{n} \textcolor{black}{|} \textcolor{black}{-}] E_{Q_2}$$

where E_{Q_1} and E_{Q_2} are the RA expressions corresponding to Q_1 and Q_2

Example

"Find the sid of each student who is only enrolled in CS courses."

```
SELECT      q1.sid
FROM        (SELECT s.sid AS sid, s.sname
            FROM Student s
            EXCEPT
            SELECT q2.sid, s.sname
            FROM (SELECT s.sid AS sid, s.sname, e.sid, e.cno
                  FROM Student s, Enroll e
                  WHERE e.sid = s.sid
                  EXCEPT
                  (SELECT s.sid, s.sname, e.sid, e.cno
                  FROM Student s, Enroll e, Course c
                  WHERE e.cno = c.cno AND c.dept = 'CS' ) q2) q1
```

Example

"Find the sid of each student who is only enrolled in CS courses."

```
SELECT q1.sid
FROM   (SELECT s.sid AS ssid, s.sname
        FROM Student s
       EXCEPT
      SELECT q2.sid, s.sname
        FROM (SELECT s.sid AS ssid, s.sname, e.sid, e.cno
              FROM Student s NATURAL JOIN Enroll e
             EXCEPT
            (SELECT s.sid, s.sname, e.sid, e.cno
              FROM Student s CROSS JOIN Enroll E
              NATURAL JOIN (SELECT c.* FROM Course c WHERE dept = 'CS') c) q2) q1
```

is translated to

$$\pi_{sid}(\pi_{sid, sname}(S) - \pi_{S.sid, sname}(\pi_{S.sid, sname, E.sid, cno}(S \bowtie J E) - \\ \pi_{S.sid, sname, E.sid, cno}(S \times E \bowtie J \sigma_{dept='CS'}(C)))$$

where S , E , and C denote Student, Enroll, and Course, respectively.

Entity-Relationship Model

E/R Diagrams

Converting E/R Diagrams to Relations

Jeff Ullman

(edited Dirk Van Gucht)

Purpose of E/R Model

- ◆ The E/R model allows us to sketch database schema designs.
 - ▷ Includes some constraints, but not operations.
- ◆ Designs are pictures called *entity-relationship diagrams*.
- ◆ **Later:** convert E/R designs to relational DB designs.

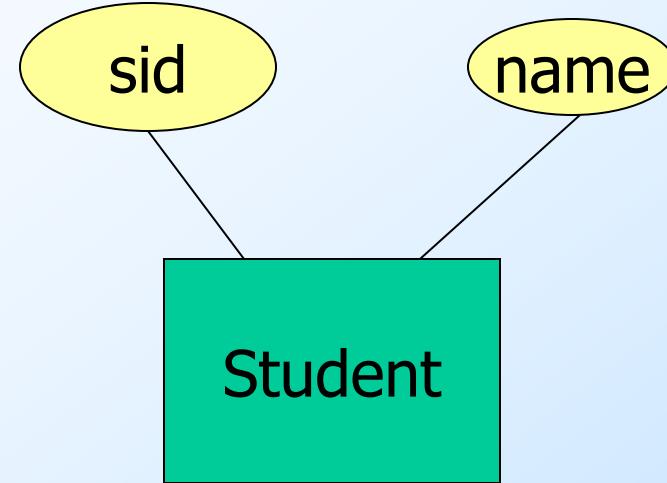
Entities and Entity Sets

- ◆ *Entity* = “thing” or object.
- ◆ *Entity set* = collection of similar entities.
 - ▶ Similar to a class in object-oriented languages.
- ◆ *Attribute* = property of an entity
 - ▶ Attributes are simple values, e.g. integers or character strings, not structs, sets, etc.

E/R Diagrams

- ◆ In an entity-relationship diagram:
 - ▷ Entity = rectangle.
 - ▷ Attribute = oval, with a line to the rectangle representing the entity

Example:

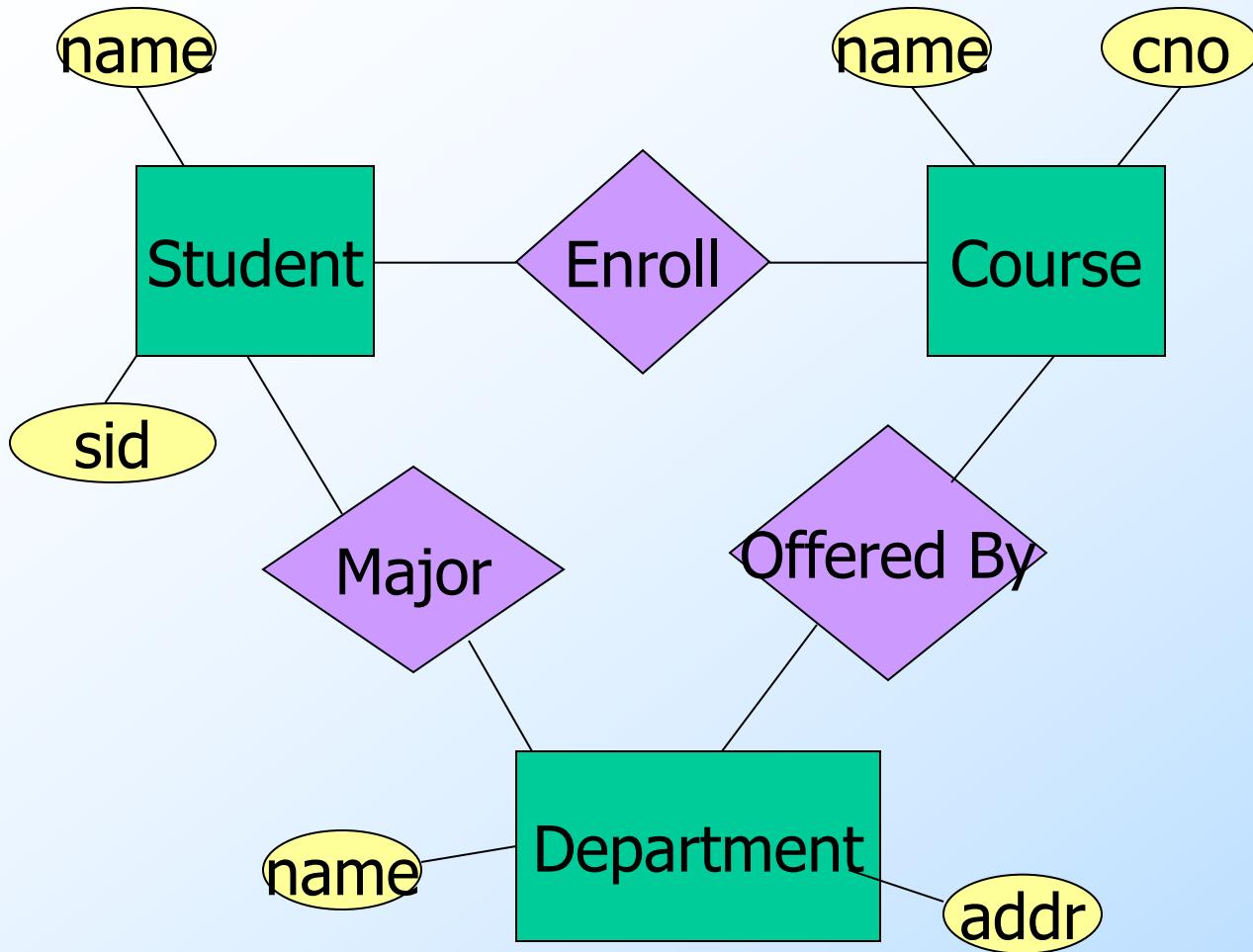


- ◆ Entity **Student** has two attributes, **sid** and **name**
- ◆ Each **Student** entity has values for these two attributes, e.g. (s1, Anna)

Relationships

- ◆ A **relationship** connects two or more entities
- ◆ It is represented by a diamond, with lines to each of the entities involved

Example: Relationships



Relationship Set

- ◆ The current “value” of an entity set is the set of entities that belong to it
 - ▷ Example: the set of all students in our database
- ◆ The “value” of a relationship is a *relationship set*, a set of tuples with one component for each related entity set

Example: Relationship Set

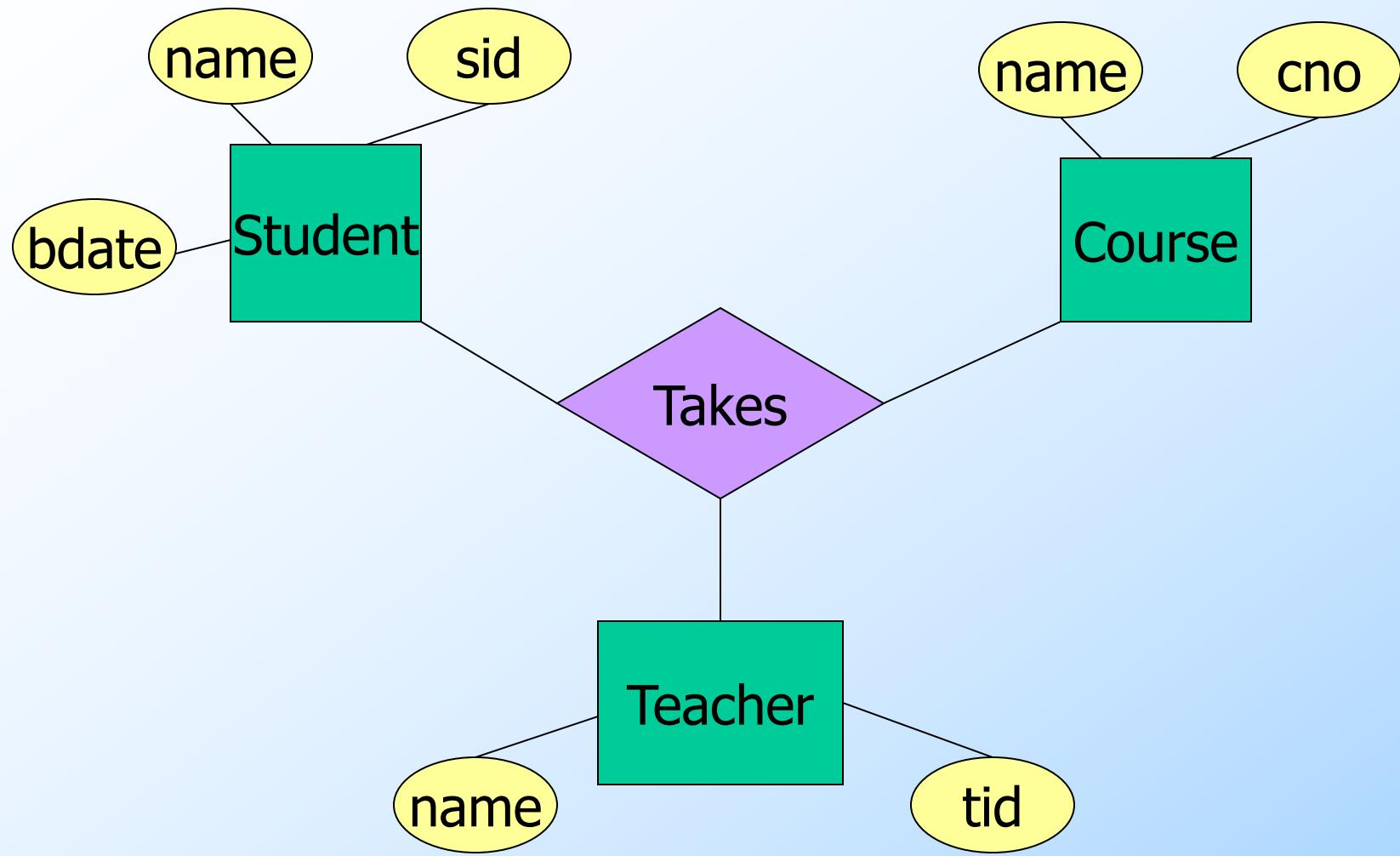
- ◆ For the relationship **Major**, we might have a relationship set like:

Student	Department
s1	CS
s1	Math
s2	Chemistry
s3	Math
s3	Physics

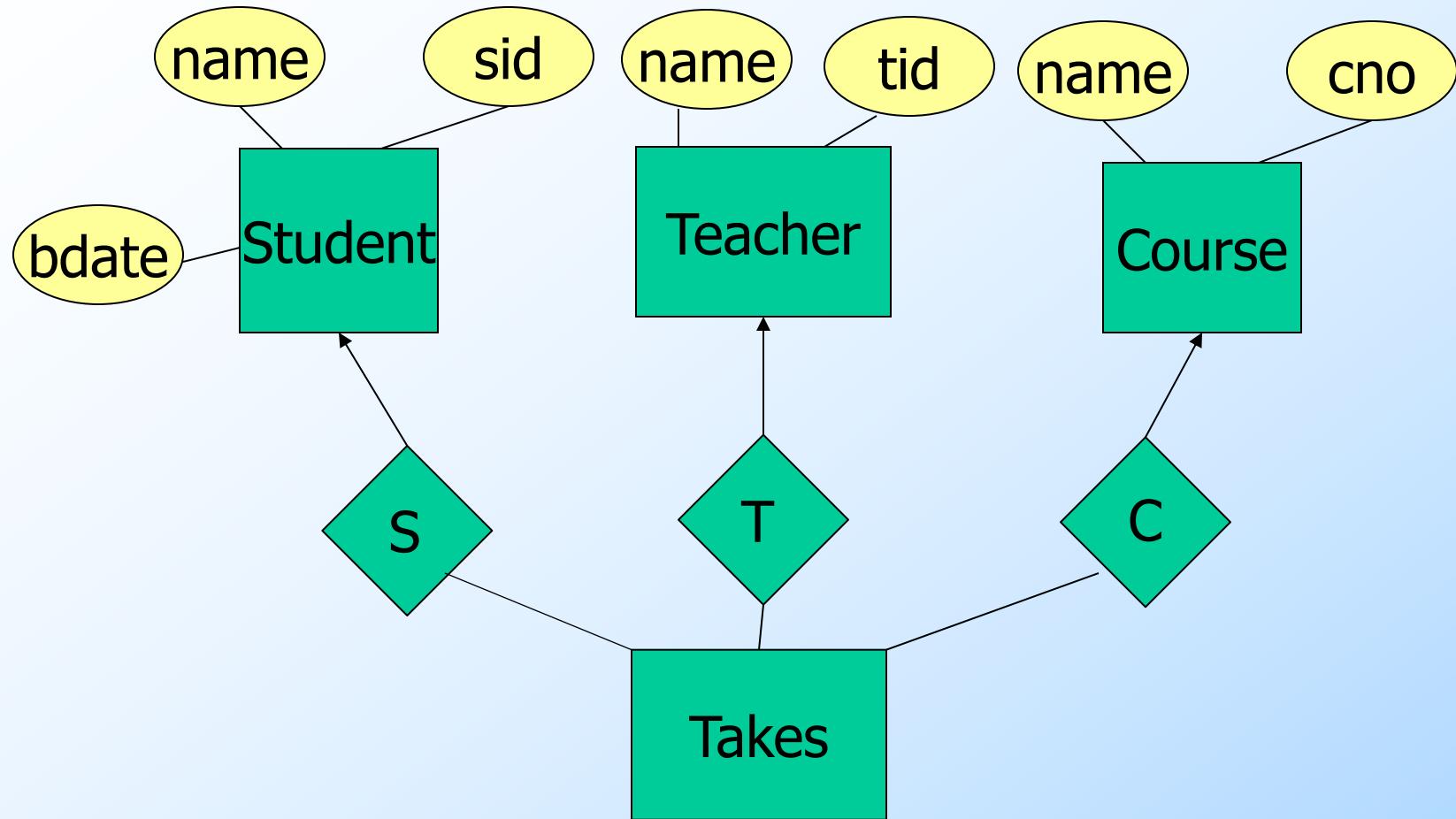
Multiway Relationships

- ◆ Sometimes, we need a relationship that connects more than two entities
- ◆ Suppose that student takes a course taught by a teacher (s,c,t)
- ◆ Three binary relationships do not allow us to precisely capture this relationship between a student, course, and teacher.
 - ▶ But a 3-way relationship would.

Example: 3-Way Relationship



Example: Representation with 3 many-to-one relationships



A Typical Relationship Set

Student	Course	Teacher
s1	c1	t1
s1	c2	t2
s2	c2	t1

Can not be captured with the three binary relationship sets

Student	Course
s1	c1
s1	c2
s2	c2

Student	Teacher
s1	t1
s1	t2
s2	t1

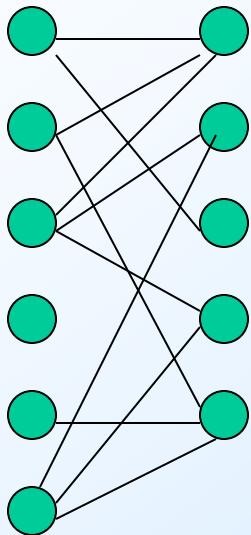
Course	Teacher
c1	t1
c2	t2
c2	t1

Notice how the triple (s1,c2,t1) is possible according to the 3 binary relations, but this tuple is not in the ternary relationship

Many-Many Relationships

- ◆ Focus: **binary** relationships, such as Enroll between **Students** and **Courses**
- ◆ In a ***many-many*** relationship, an entity of either set can be connected to many entities of the other set
 - ▶ E.g., a student may enroll in many courses; a course may enroll many students.

In Pictures:

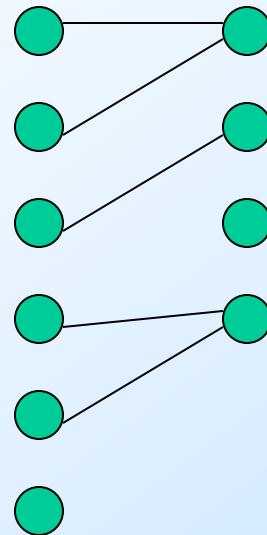


many-many

Many-One Relationships

- ◆ Some binary relationships are *many-one* from one entity set to another
- ◆ Each entity of the first set is connected to at most one entity of the second set
- ◆ But an entity of the second set can be connected to zero, one, or many entities of the first set
- ◆ A many-one relationship is a **function**

In Pictures:



many-one

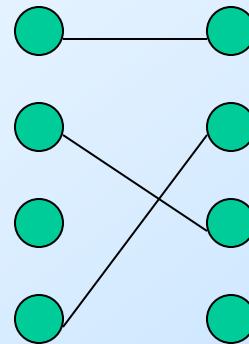
Example: Many-One Relationship

- ◆ Offered By, from Courses to Departments is many-one
- ◆ A course is offered by at most one department
- ◆ But a department can offer any number of students, including zero

One-One Relationships

- ◆ In a *one-one relationship*, each entity of either entity set is related to at most one entity of the other set
- ◆ Example: Relationship **Chair Person** between entity sets **Teachers** and **Department**
 - A department cannot be chaired by more than one teacher, and no teacher can chair more than one department

In Pictures:

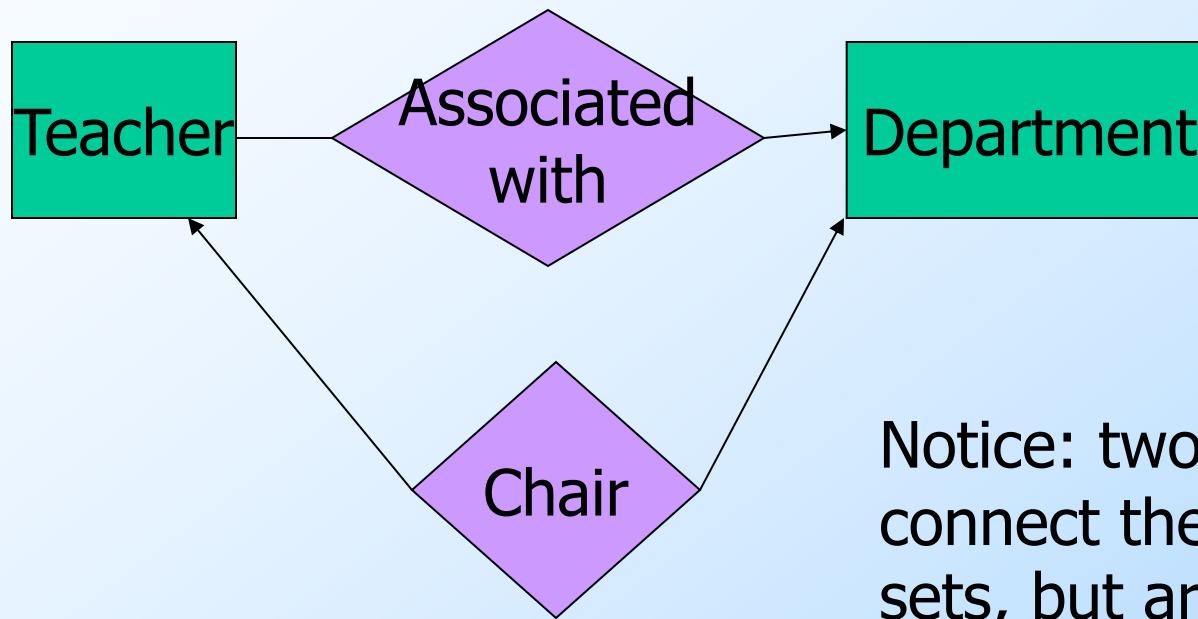


one-one

Representing “Multiplicity”

- ◆ Show a many-one relationship by an arrow entering the “one” side
 - ▷ Remember: Like a function
- ◆ Show a one-one relationship by arrows entering both entity sets
- ◆ **Rounded arrow** = “exactly one,” i.e., each entity of the first set is related to exactly one entity of the target set

Example: Many-One Relationship

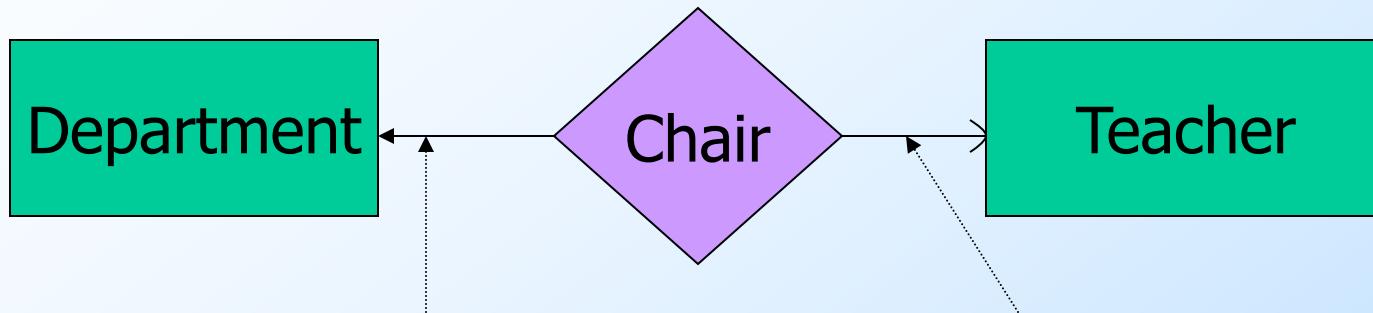


Notice: two relationships connect the same entity sets, but are different.

Example: One-One Relationship

- ◆ Consider Chair between Teachers and Departments.
- ◆ Some teachers are not the chair of any department, so a rounded arrow to department would be inappropriate
- ◆ But a department must have chair person.

In the E/R Diagram



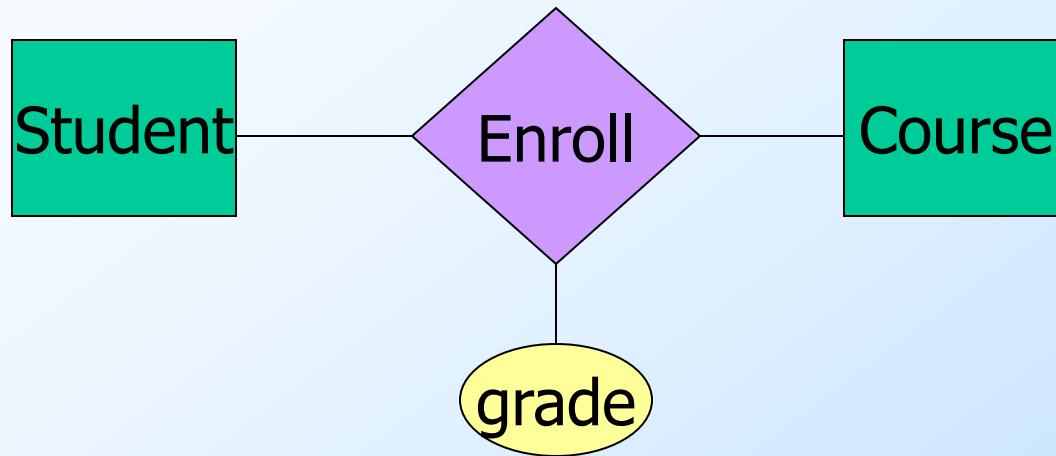
A department has exactly one chair.

A teacher is the chair of at most one (zero or one) department.

Attributes on Relationships

- ◆ Sometimes it is useful to attach an attribute to a relationship
- ◆ Think of this attribute as a property of each tuple (relationship) in the relationship set

Example: Attribute on Relationship



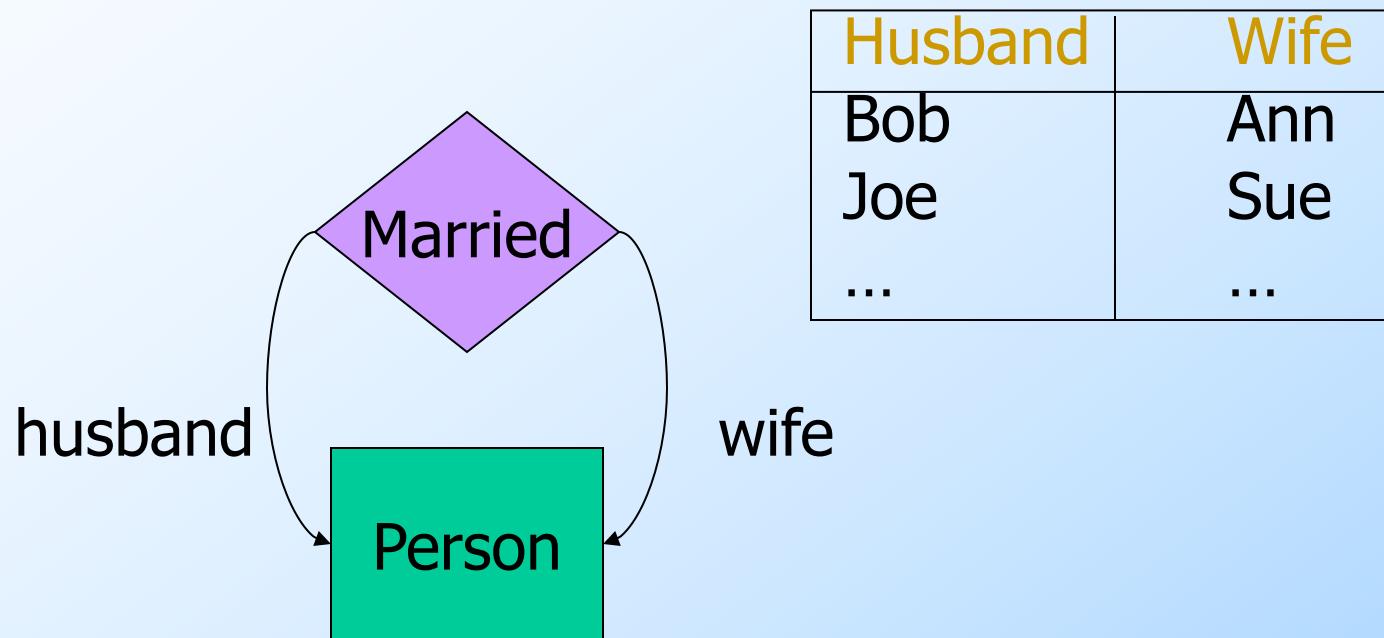
Grade is a function of both the student and the course, not of one alone

Roles

- ◆ Sometimes an entity set appears more than once in a relationship
- ◆ Label the edges between the relationship and the entity set with names called *roles*

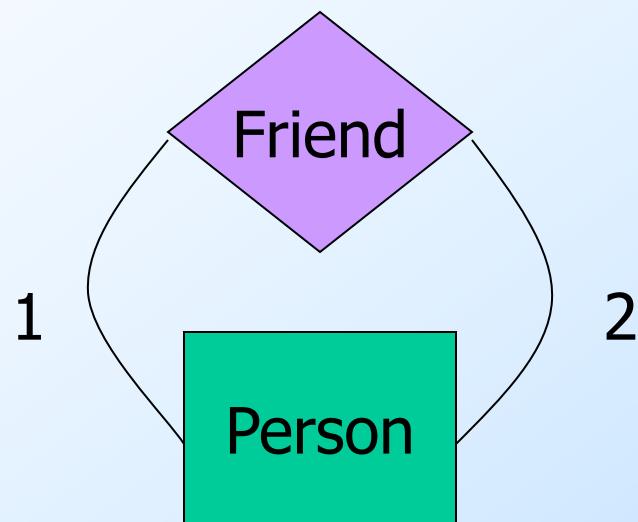
Example: Roles

Relationship Set



Example: Roles

Relationship Set



Friend1	Friend2
Bob	Ann
Joe	Sue
Ann	Bob
Joe	Moe
...	...

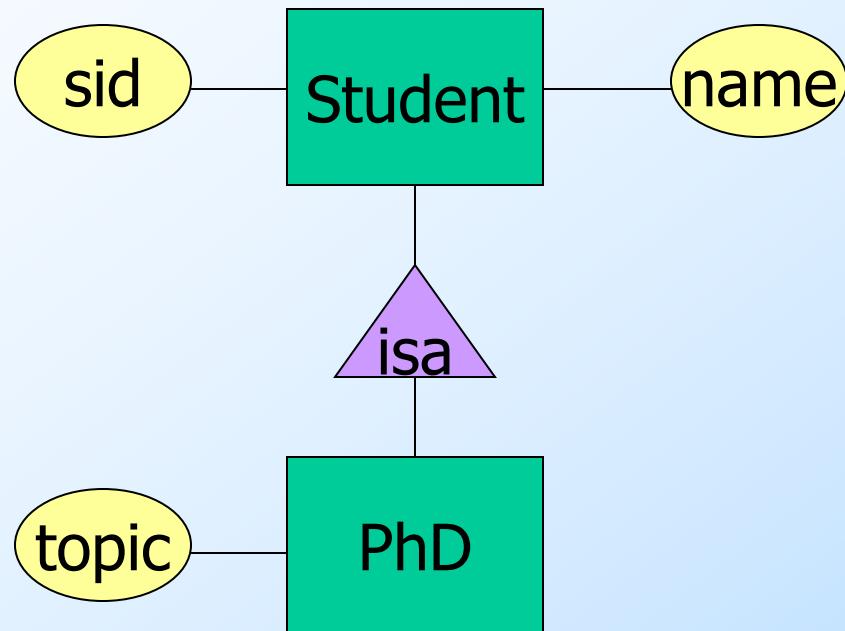
Subclasses

- ◆ *Subclass* = special case = fewer entities
= more properties = more relationships
- ◆ **Example:** PhD students are a kind of student.
 - ▶ Not every student is a PhD student, but some are.
 - ▶ Let us suppose that in addition to all these *properties* (attributes and relationships), a PhD student also has the attribute **topic**

Subclasses in E/R Diagrams

- ◆ Assume subclasses form a tree.
 - ▷ I.e., no multiple inheritance.
- ◆ Isa triangles indicate the subclass relationship.
 - ▷ Point to the superclass.

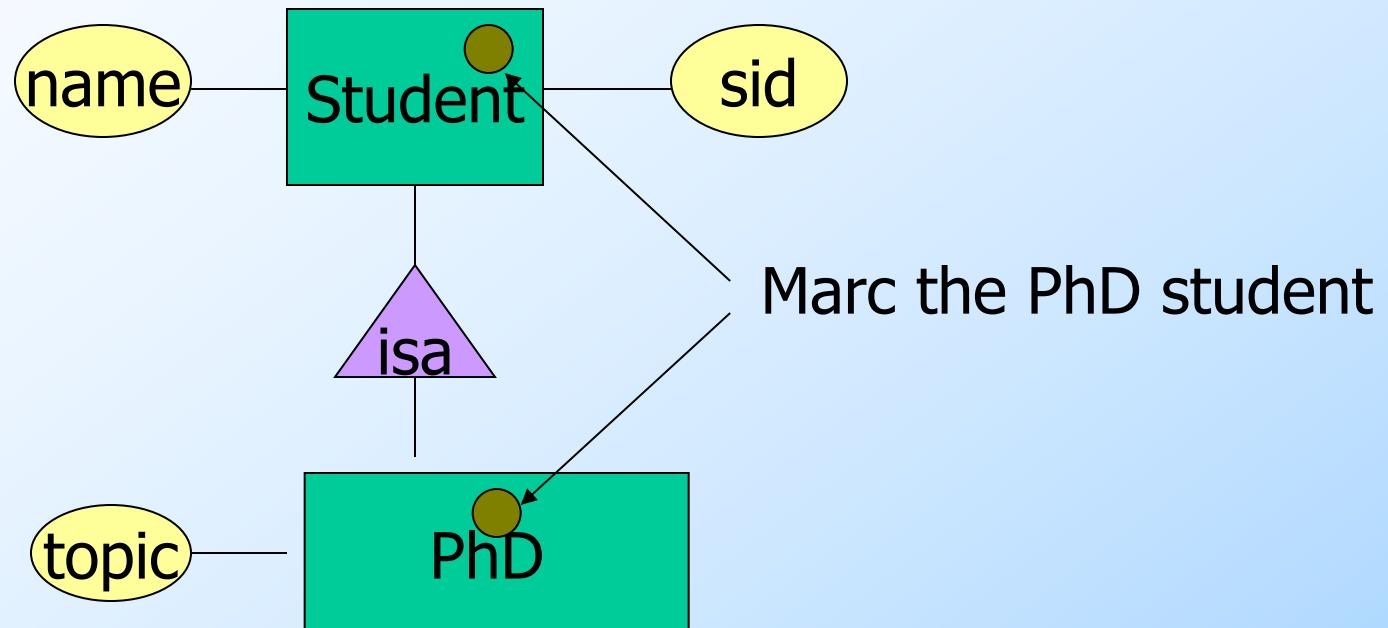
Example: Subclasses



E/R Vs. Object-Oriented Subclasses

- ◆ In OO, objects are in one class only.
 - ▶ Subclasses inherit from superclasses.
- ◆ In contrast, E/R entities have *representatives* in all subclasses to which they belong.
 - ▶ **Rule:** if entity e is represented in a subclass, then e is represented in the superclass (and recursively up the tree).

Example: Representatives of Entities



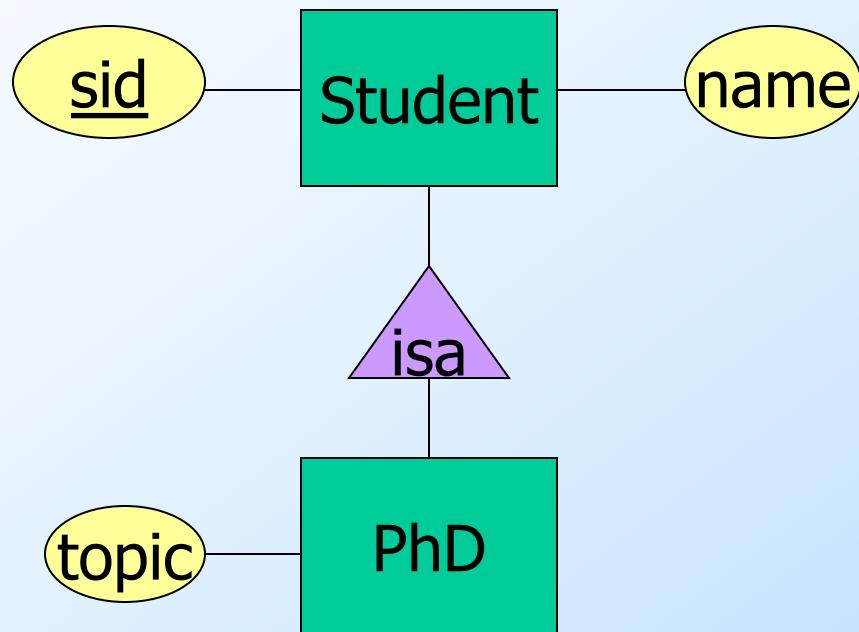
Keys

- ◆ A *key* is a set of attributes for one entity set such that no two entities in this set agree on all the attributes of the key
 - ▶ It is allowed for two entities to agree on some, but not all, of the key attributes.
- ◆ We must designate a key for every entity set

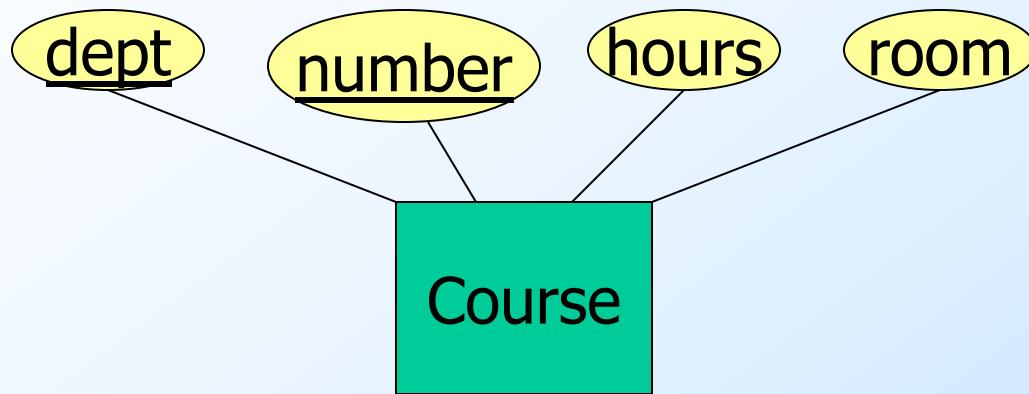
Keys in E/R Diagrams

- ◆ Underline the key attribute(s)
- ◆ In an Isa hierarchy, only the root entity set has a key, and it must serve as the key for all entities in the hierarchy

Example: name is Key for Student



Example: a Multi-attribute Key



- Note that **hours** and **room** could also serve as a key, but we must select only one key.

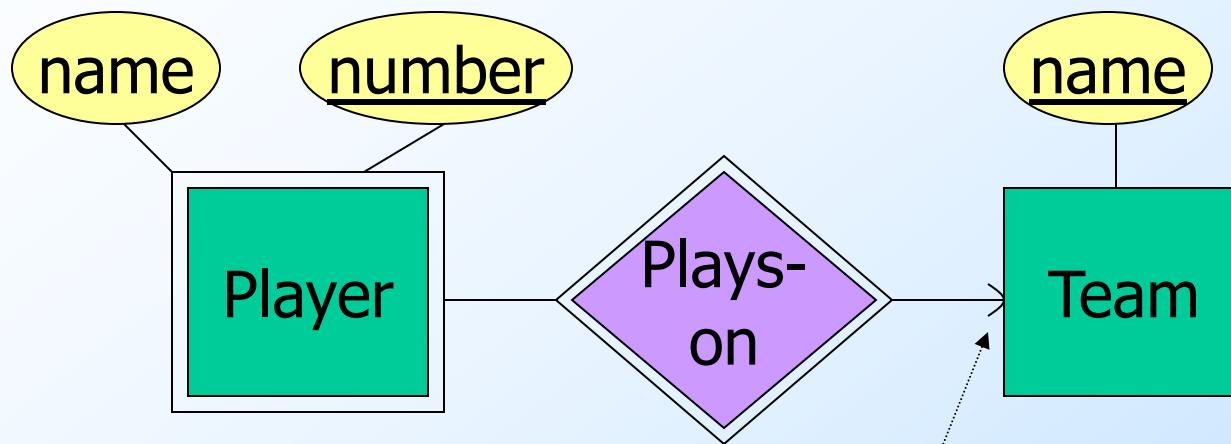
Weak Entity Sets

- ◆ Occasionally, entities of an entity set need “help” to identify them uniquely.
- ◆ Entity set E is said to be *weak* if in order to identify entities of E uniquely, we need to follow one or more many-one relationships from E and include the key of the related entities from the connected entity sets.

Example: Weak Entity Set

- ◆ **name** is almost a key for football players, but there might be two with the same name
- ◆ **number** is certainly not a key, since players on two teams could have the same number.
- ◆ But **number**, together with the team **name** related to the player by **Plays-on** should be unique.

In E/R Diagrams



Note: must be rounded
because each player needs
a team to help with the key.

- Double diamond for *supporting* many-one relationship.
- Double rectangle for the weak entity set.

Weak Entity-Set Rules

- ◆ A weak entity set has one or more many-one relationships to other (supporting) entity sets
 - ▷ Not every many-one relationship from a weak entity set need be supporting
 - ▷ But supporting relationships must have a rounded arrow (entity at the “one” end is guaranteed).

Weak Entity-Set Rules – (2)

- ◆ The key for a weak entity set is its own underlined attributes and the keys for the supporting entity sets.
 - ▷ E.g., (player) **number** and (team) **name** is a key for **Players** in the previous example.

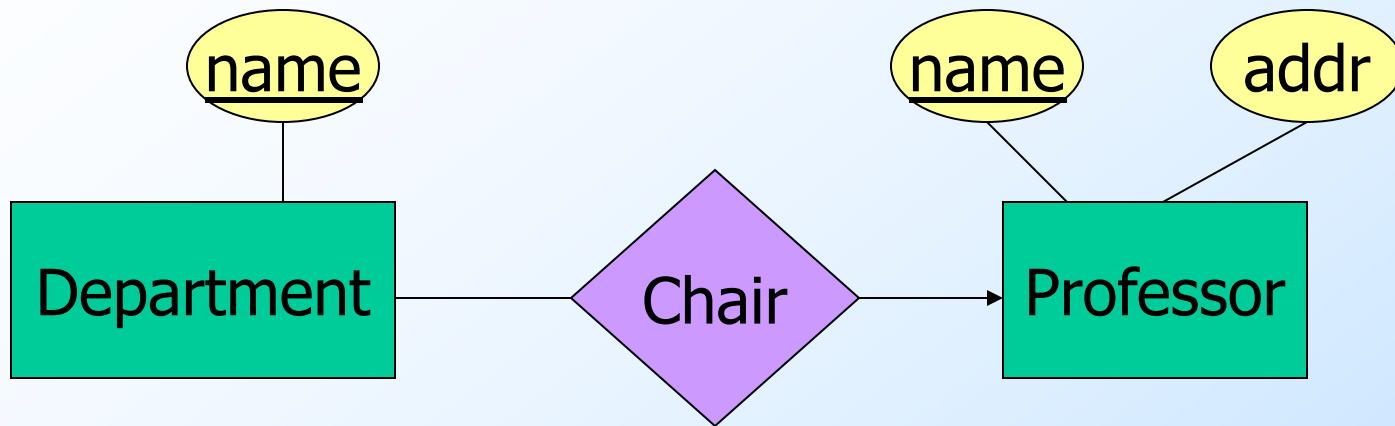
Design Techniques

1. Avoid redundancy
2. Limit the use of weak entity sets
3. Don't use an entity set when an attribute will do

Avoiding Redundancy

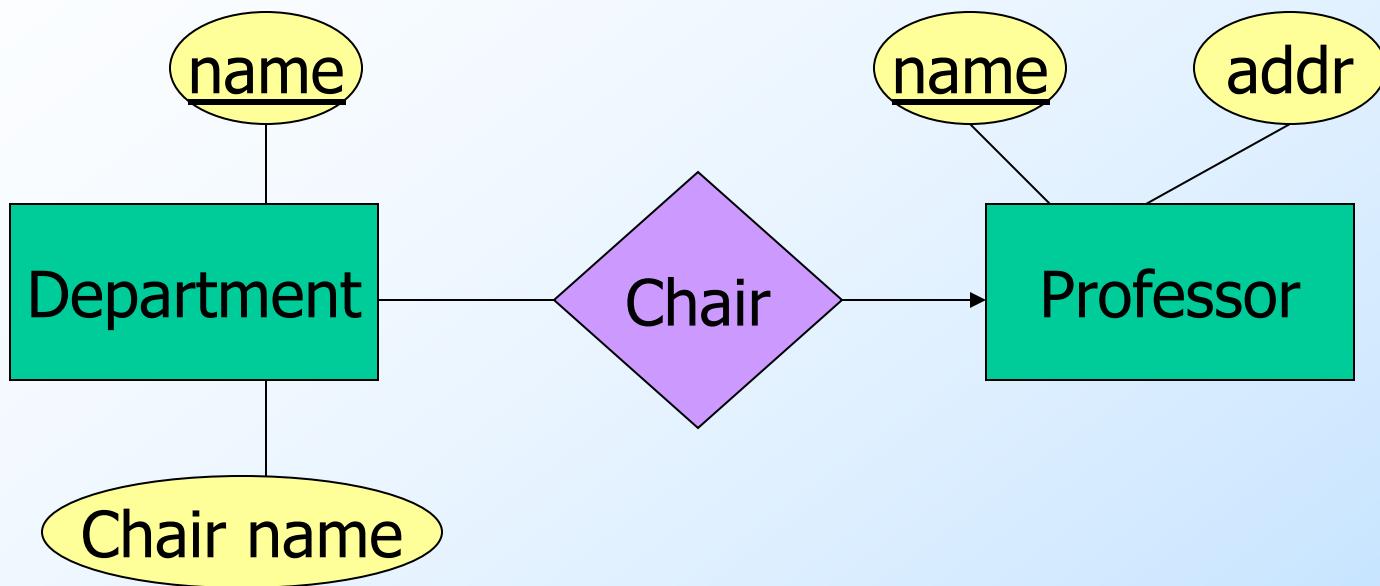
- ◆ *Redundancy* = saying the same thing in two (or more) different ways
- ◆ Wastes space and (more importantly) encourages inconsistency
 - ▷ Two representations of the same fact become inconsistent if we change one and forget to change the other.

Example: Good



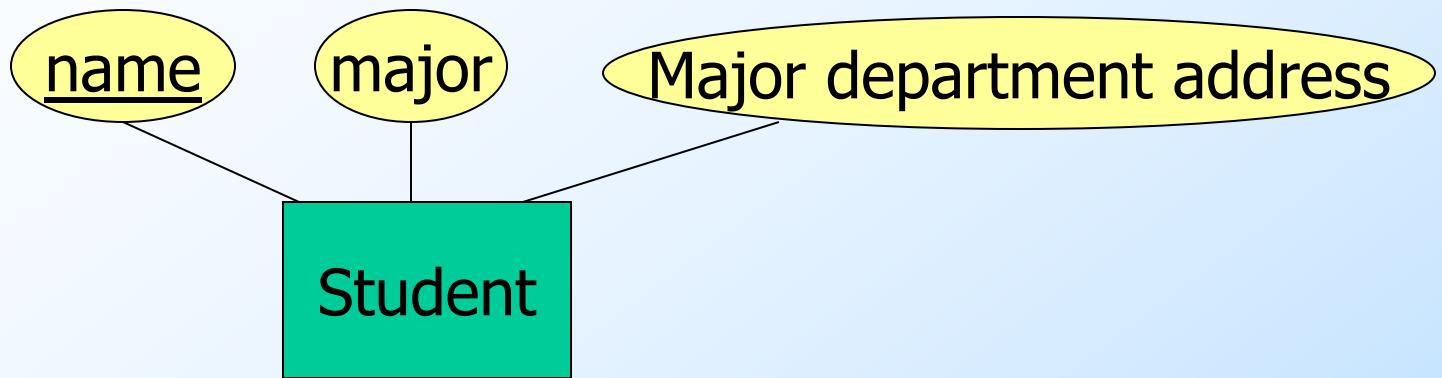
This design gives the address of each professor exactly once

Example: Bad



This design states the chair of a department twice: as an attribute and as a related entity.

Example: Bad

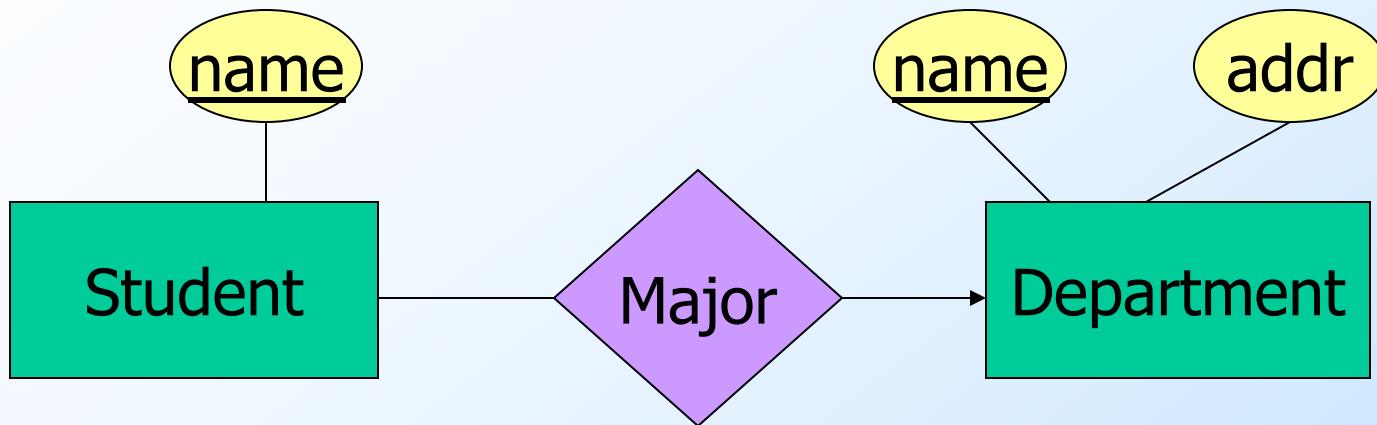


This design repeats the major's department address once for each student and loses the major department address if there are temporarily no students for a major

Entity Sets Versus Attributes

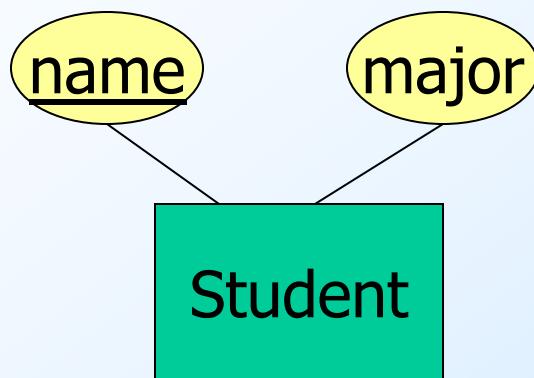
- ◆ An entity set should satisfy at least one of the following conditions:
 - ▷ It is more than the name of something; it has at least one nonkey attribute.
or
 - ▷ It is the “many” in a many-one or many-many relationship.

Example: Good



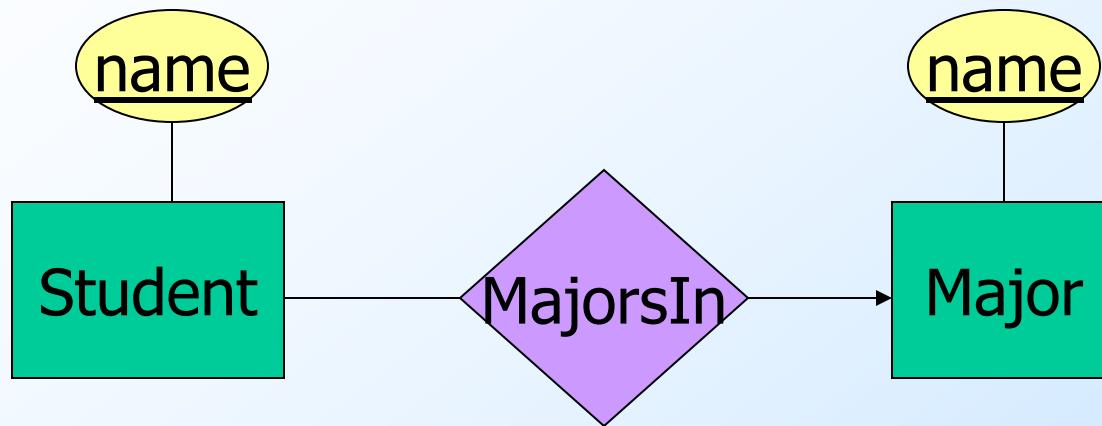
- **Department** deserves to be an entity set because of the nonkey attribute **addr**.
- **Student** deserves to be an entity set because it is the “many” of the many-one relationship **Major**.

Example: Good



There is no need to make the major an entity set, because we record nothing about majors besides their name

Example: Bad



Since the major is nothing but a name, and is not at the “many” end of any relationship, it should not be an entity set.

Don't Overuse Weak Entity Sets

- ◆ Beginning database designers often doubt that anything could be a key by itself
 - ▶ They make all entity sets weak, supported by all other entity sets to which they are linked.
- ◆ In reality, we usually create unique ID's for entity sets
 - ▶ Examples include social-security numbers, automobile VIN's etc.

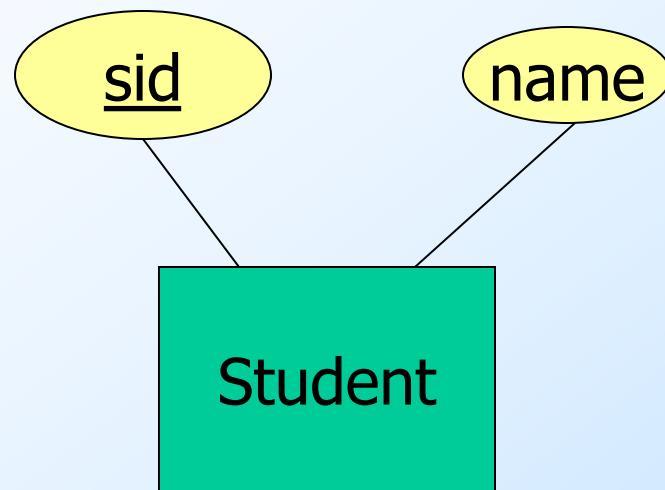
When Do We Need Weak Entity Sets?

- ◆ The usual reason is that there is no global authority capable of creating unique ID's
- ◆ **Example:** it is unlikely that there could be an agreement to assign unique player numbers across all football teams in the world

From E/R Diagrams to Relations

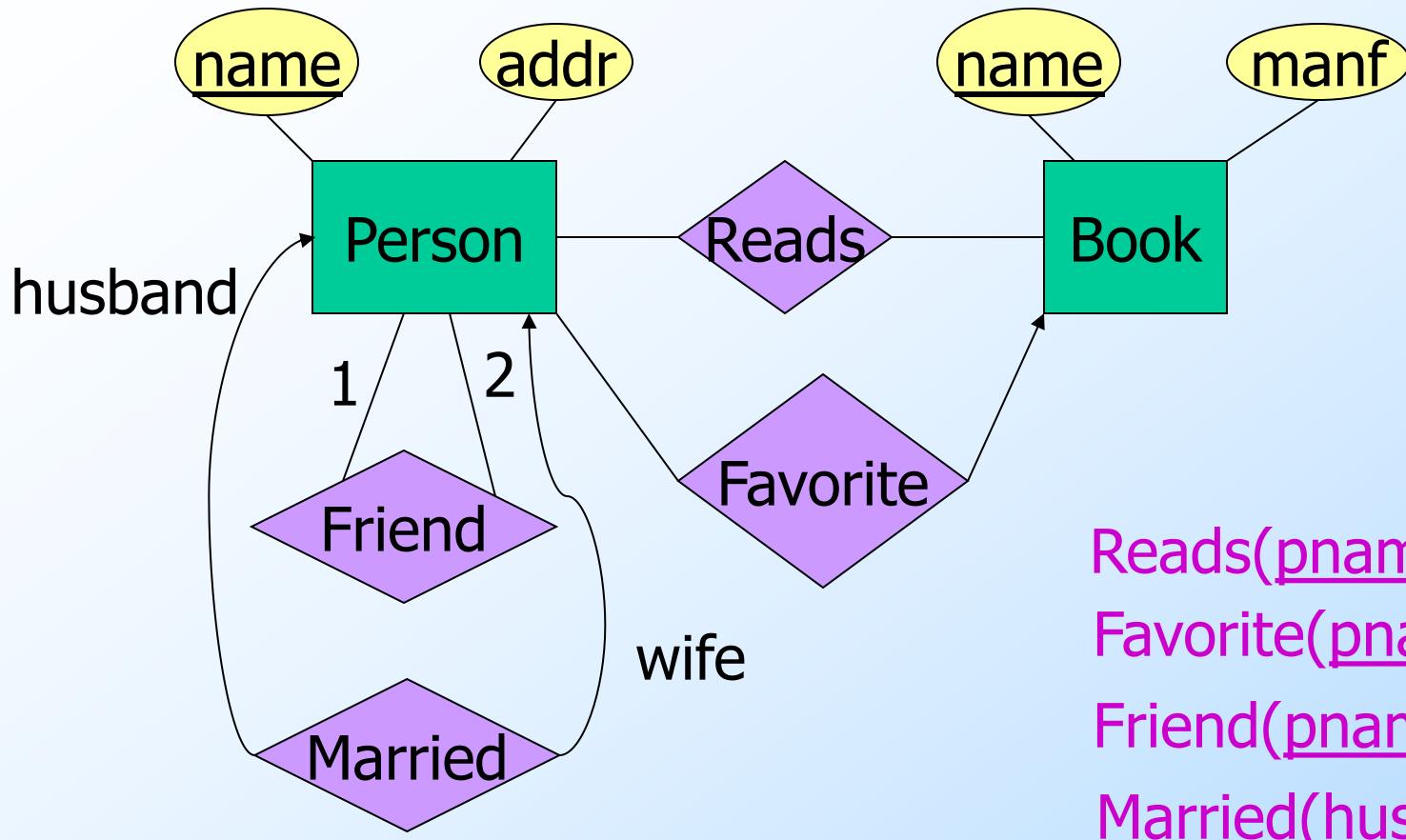
- ◆ Entity set -> relation.
 - ▷ Attributes -> attributes.
- ◆ Relationships -> relations whose attributes are only:
 - ▷ The keys of the connected entity sets.
 - ▷ Attributes of the relationship itself.

Entity Set -> Relation



Relation: **Student(sid, name)**

Relationship -> Relation



Combining Relations

- ◆ OK to combine into one relation:
 1. The relation for an entity-set E
 2. The relations for many-one relationships of which E is the “many.”
- ◆ Example: Person(pname, addr) and Favorite(pname, bname) combine to make Person1(pname, addr, favBook).

Risk with Many-Many Relationships

- ◆ Combining Student with Course would be a mistake. It leads to redundancy, as:

name	addr	course
Sally	123 Maple	Databases
Sally	123 Maple	Compilers

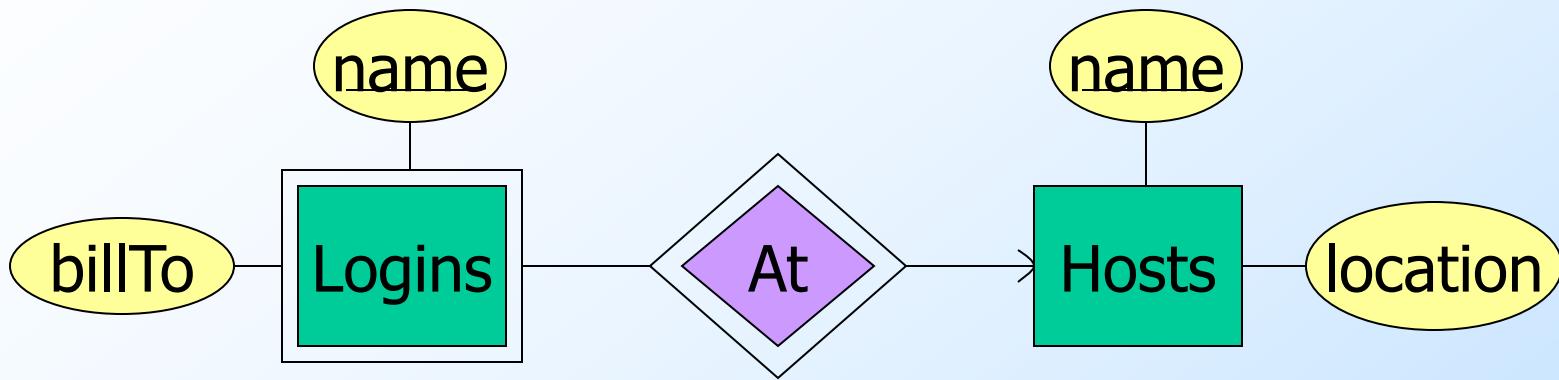
Redundancy



Handling Weak Entity Sets

- ◆ Relation for a weak entity set must include attributes for its complete key (including those belonging to other entity sets), as well as its own, nonkey attributes.

Example: Weak Entity Set -> Relation

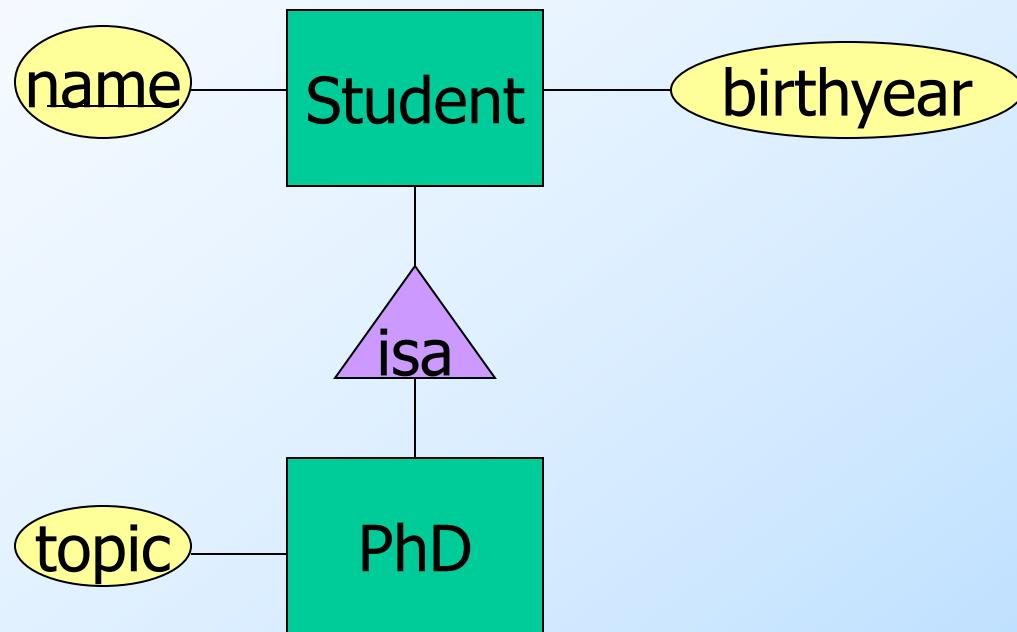


Hosts(hostName, location)
Logins(loginName, hostName, billTo)

Subclasses: Three Approaches

1. *Object-oriented* : One relation per subset of subclasses, with all relevant attributes.
2. *Use nulls* : One relation; entities have NULL in attributes that don't belong to them.
3. *E/R style* : One relation for each subclass:
 - ▶ Key attribute(s).
 - ▶ Attributes of that subclass.

Example: Subclass -> Relations



Object-Oriented

<u>name</u>	birthyear
Ann	1999

Student

<u>name</u>	birthyear	topic
Ellen	1995	PL

PhD

Good for queries like “find the topics of PhD’s born after 1990”

E/R Style

<u>name</u>	<u>birthyear</u>
Ann	1999
Ellen	1995

Student

<u>name</u>	<u>topic</u>
Ellen	PL

PhD

Good for queries like
“find all student (including
PhDs) born in 1995.”

Using Nulls

name	birthyear	topic
Ann	1999	NULL
Ellen	1995	PL

Student

Saves space unless there are *lots* of attributes that are usually NULL.

In SQL (relationships)

- ◆ Consider student, course, enroll
- ◆ Maintain referential integrity

```
create table student (sid integer PRIMARY KEY,  
                      name text);
```

```
create table course (cno integer PRIMARY KEY,  
                     name text);
```

```
create table enroll (sid integer REFERENCES student(sid),  
                    cno integer REFERENCES course(cno),  
                    grade VARCHAR(2),  
                    primary key(sid,cno))
```

In SQL (subclasses) Object-Oriented style

- ◆ Consider PhD example

```
CREATE TABLE Student(sid PRIMARY KEY,  
                    name text,  
                    birthyear integer);
```

```
CREATE TABLE PhD(sid PRIMARY KEY,  
                 name text,  
                 birthyear integer,  
                 topic integer);
```

- ◆ Notice that there is no foreign key in the PhD table.

In SQL (subclasses) E/R style

- ◆ Consider student, PhD

```
create table Student (sid integer PRIMARY KEY,  
                      name text,  
                      birthyear integer);
```

```
create table PhD (sid integer REFERENCES Student(sid),  
                  topic text);
```

In SQL (subclasses) With NULLS

- ◆ Consider PhD example

```
CREATE TABLE Student (sid integer PRIMARY KEY,  
                      name text,  
                      birthyear integer,  
                      phd_topic text);
```

The Object-Relational Database Model- an Entry in noSQL database models

Dirk Van Gucht¹

¹Indiana University

March 9, 2019

Motivation

- In the relational database model, each relation has a schema of attributes with **atomic domains** such as booleans, numbers, text strings etc
- In the **object-relational model**, and in various **noSQL data models**, attributes in the schema the database may also have domains that consists of **complex objects** such as arrays, sets, bags, objects of composite types, relations, **JSON** objects, **XML** documents etc
- Therefore we need mechanisms to
 - ❶ define relations/databases with attributes of complex-object types; and
 - ❷ search and manipulate such relations/databases

Set and bag types as arrays

- In general, complex-object types can be recursively defined in terms of atomic types, composite types, array types, etc
- The main focus of this lecture will be on **array types**
- In particular, we will show how array types can be used to model **bag** and **set** types
- We will then show how operations on arrays allow us to model operations on bags and sets
- **PostgreSQL** is an excellent system to consider the issues since it is an object-relational database system
- Many of the concepts discussed here can also be found in the **NoSQL MongoDB** system as well as in the **MapReduce** framework and its derivatives

Arrays and the array constructor operation

- In SQL,

`‘{7, 4, 4, 3, 2}’::int[]`

denotes the array [7, 4, 3, 3, 2] of type `int[]`

- Its first, third, and fifth component values are obtained as follows:

Array component	Value
<code>(‘{7, 4, 4, 3, 2}’::int[])[1]</code>	7
<code>(‘{7, 4, 4, 3, 2}’::int[])[3]</code>	4
<code>(‘{7, 4, 4, 3, 2}’::int[])[5]</code>	2

- In SQL,

`‘{“C”, “John”, “Anna”, “12”}’::text[]`

denotes the array [‘C’, ‘John’, ‘Anna’, ‘12’] of type `text[]`

- Elements of an array must all be of the same type

Arrays and the array constructor operation (alternative syntax)

- In SQL, the following all denote the same array of integers

[7, 4, 3, 3, 2]

{7, 4, 4, 3, 2}::int[]
ARRAY[7,4,4,3,2]::int[]
ARRAY[7,4,4,3,2]

- The third component value is obtained as follows

Array component	Value
({7, 4, 4, 3, 2}::int[])[3]	4
(ARRAY[7,4,4,3,2]::int[])[3]	4
(ARRAY[7,4,4,3,2])[3]	4

Modeling bags and sets with arrays

- The array

$\text{ARRAY}[7,4,4,3,2]$

represent (models) the bag

$\{2, 3, 4, 4, 7\}$

and the set

$\{2, 3, 4, 7\}$

- Recall that an array orders its elements but a bag or a set does not
- The arrays $\text{ARRAY}[7,4,4,3,2]$ and $\text{ARRAY}[2,4,3,7,4]$ are different but they both represent the same bag and the same set
- The empty array ' $\{\}$ ' or $\text{ARRAY}[]$ models the empty set $\{\}$ (i.e., \emptyset)

ARRAY construction from a unary SQL query

- The ARRAY constructor operation can be applied to any SQL query that returns a **unary** relation
- It constructs an array of the elements of that relation

A
x
1
3
2
2

`SELECT ARRAY(SELECT x FROM A)`

→

array
{1, 3, 2, 2}

A
x
1
3
2
2

`SELECT ARRAY(SELECT DISTINCT x FROM A ORDER BY x)`

→

array
{1, 2, 3}

ARRAY construction from a SQL query with ROW construction

- The ARRAY constructor operation can be applied to any SQL query
- But, the tuples returned by the query must be **packed** to be of a composite (row) type by the **ROW** constructor operation

A	
x	y
1	a
1	b
2	a

`SELECT ARRAY(SELECT ROW(x,y) FROM A)`

`SELECT ARRAY(SELECT (x,y) FROM A)`

array

`{(1, a), (1, b), (2, a)}`

Example: A documents relation

- We may wish to maintain a database of documents and the words they contain
- We can define a relation with attributes of atomic domain to store such documents

```
CREATE TABLE documentWord (doc text, word text);
```

- A pair (d, w) in `documentWord` specifies that document d contains the word w

Example: A documents relation

- The `documentWord` relation may look as follow:

`documentWord`

doc	word
d1	A
d1	B
d1	C
d2	B
d2	C
d2	D
d3	A
d3	E
d4	B
d4	B
d4	A
d4	D
d5	E
d5	F
d6	A
d6	D
d6	G
d7	C
d7	B
d7	A
d8	B
d8	A

Example: The documents relation as a complex-objects relation

- We could consider a more **natural representation** of this data by having a relation **documents** of pairs (doc, words) where we pair each document with its set (bag) of words

documents	
L	
doc	words
d1	{A, B, C}
d2	{B, C, D}
d3	{A, E}
d4	{B, B, A, D}
d5	{E, F}
d6	{A, D, G}
d7	{C, B, A}
d8	{B, A}

- Such a relation is called a **complex-objects relation**
- We will see how the **ARRAY** type can be used to model such complex-objects relations
- We will then discuss how such relations can be manipulated and queried

The **ARRAY** type

- SQL has the array type
 - for example, the array type `text[]` declares an array of text;
 - `int[]` declares an array of int;
- SQL permits the use of these types in the definition of complex-object relations. For the `documents` relation, we can use the declaration

```
CREATE TABLE documents (doc text, words text[]);
```

- Such a table can be populated using insert statements such as

...

```
INSERT INTO documents VALUES ('d6', '{"A","D","G"}');
```

...

Querying the documents relation

- Next consider the query

```
SELECT d.doc, d.words FROM documents d
```

- This query returns the contents of the documents relation
- The result would be

doc	words
d1	{A, B, C}
d2	{B, C, D}
d3	{A, E}
d4	{B, B, A, D}
d5	{E, F}
d6	{A, D, G}
d7	{C, B, A}
d8	{B, A}

Set and Bags as Unordered Array

- We will use arrays to represent sets (or bags).
- We must therefore restrict the predicates and operations we define on arrays to be **independent** of the order in which the elements appear in the arrays
- The following are such predicates and operations

$a \in A$	a is a member (element) of set A
$a \notin A$	a is not a member (element) of set A
$A \cap B \neq \emptyset$	set A and set B overlap
$A \subseteq B$	set A is a subset of set B
$A \supseteq B$	set A is a superset of set B
$A = \emptyset$	set A is empty
$ A $	denotes the cardinality (size) of set A
$A \cup B, A \cap B, A - B$	union , intersection , difference of sets A and B

Checking for Set Membership ($a \in A$)

- In SQL this can be done using the $= \text{SOME}$ predicate
- “Find the documents that contain the word ‘D’ ”

```
SELECT d.doc, d.words
FROM   documents d
WHERE  'D' = SOME(d.words)
```

doc	words
d2	{B, C, D}
d4	{B, B, A, D}
d6	{A, D, G}

Checking for Set Non-Membership ($a \notin A$)

- “Find the documents that do **not** contain the word ‘D’ ”
- For this we can use the **<> ALL** predicate.

```
SELECT d.doc, d.words  
FROM   documents d  
WHERE  'D' <> ALL(d.words)
```

doc	words
d1	{A, B, C}
d3	{A, E}
d5	{E, F}
d7	{C, B, A}
d8	{B, A}

The `isIn` set-membership function

- For convenience, we define a polymorphic function `isIn` for the set-membership predicate:

```
CREATE FUNCTION isIn (x anyelement, A anyarray)
RETURNS boolean AS
$$
SELECT x = SOME(A);
$$ LANGUAGE SQL;
```

- We can now write the query “Find the documents that contain the word ‘A’ but not the word ‘D’ ” as follows:

```
SELECT d.doc, d.words
FROM   documents d
WHERE  isIn('A',d.words) and not(isIn('D',d.words))
```

Checking for Overlap of Sets ($A \cap B \neq \emptyset$)

- We may wish to check if sets overlap, i.e., if they have a non-empty intersection
- This can be done using the `&&` predicate.
- “Find the documents whose sets of words overlap with the set of words $\{B, C\}$.”

```
SELECT d.doc, d.words  
FROM   documents d  
WHERE  d.words && {'"B","C"'}'
```

doc	words
d1	{A, B, C}
d2	{B, C, D}
d4	{B, B, A, D}
d7	{C, B, A}
d8	{B, A}

Checking for Disjoint (Non-overlapping) Sets ($A \cap B = \emptyset$)

- “Find the pairs of documents that do not have words in common.”

```
SELECT d1.doc AS doc1, d2.doc AS doc2,  
       d1.words AS words1, d2.words AS words2  
  FROM documents d1, documents d2  
 WHERE NOT( d1.words && d2.words )
```

doc1	doc2	words1	words2
d1	d5	{A, B, C}	{E, F}
d2	d3	{B, C, D}	{A, E}
d2	d5	{B, C, D}	{E, F}
d3	d2	{A, E}	{B, C, D}
d4	d5	{B, B, A, D}	{E, F}
d5	d1	{E, F}	{A, B, C}
d5	d2	{E, F}	{B, C, D}
d5	d4	{E, F}	{B, B, A, D}
d5	d6	{E, F}	{A, D, G}
d5	d7	{E, F}	{C, B, A}
d5	d8	{E, F}	{B, A}
d6	d5	{A, D, G}	{E, F}
d7	d5	{C, B, A}	{E, F}
d8	d5	{B, A}	{E, F}

Checking for Set Containment (subset) ($A \subseteq B$)

- We may wish to check if a set is a **subset** of another set
- This can be done using the **<@** set-containment predicate
- “Find the documents that contain the words ‘A’ and ‘B’ ”

```
SELECT    d.doc, d.words
FROM      documents d
WHERE     {'A", "B"} <@ d.words
```

doc	words1
d1	{A, B, C}
d4	{B, B, A, D}
d7	{C, B, A}
d8	{B, A}

Checking for Set Containment (subset)

- “Find the pairs of different documents d_1, d_2 such that all words in d_1 also occur as words in d_2 .”

```
SELECT  d1.doc AS doc1, d2.doc AS doc2,  
        d1.words AS words1, d2.words AS words2  
FROM    documents d1, documents d2  
WHERE   d1.words <@ d2.words AND  
        d1.doc <> d2.doc
```

doc1	doc2	words1	words2
d1	d7	{A, B, C}	{C, B, A}
d7	d1	{C, B, A}	{A, B, C}
d8	d1	{B, A}	{A, B, C}
d8	d4	{B, A}	{B, B, A, D}
d8	d7	{B, A}	{C, B, A}

Checking for Set Equality ($A = B$)

- We may wish to check if two sets are **equal**
- This can again be done using the **<@** set-containment predicate
- “Find the pairs of different documents d_1, d_2 that have the same words.”

```
SELECT  d1.doc AS doc1, d2.doc AS doc2,  
        d1.words AS words1, d2.words AS words2  
FROM    documents d1, documents d2  
WHERE   d1.words <@ d2.words AND  
        d2.words <@ d1.words AND  
        d1.doc <> d2.doc
```

doc1	doc2	words1	words2
d1	d7	{A, B, C}	{C, B, A}
d7	d1	{C, B, A}	{A, B, C}

Caveat: Do not use ARRAY equality = to test set-equality

- Consider the ARRAY equality predicate ‘=’
- This predicate checks if two arrays are the same, i.e., they are equal **component by component**
- So ‘=’ is an **order-dependent** predicate and should therefore not be used in our context of set predicates and operations

```
SELECT  d1.doc AS doc1, d2.doc AS doc2,  
        d1.words AS words1, d2.words AS words2  
FROM    documents d1, documents d2  
WHERE   d1.words = d2.words AND  
        d1.doc <> d2.doc
```

will return the empty set

Checking for Set Emptyness ($A = \emptyset$)

- “Find the documents that contain no words.”

```
SELECT    d.doc, d.words
FROM      documents d
WHERE     d.words <@ '{}'
```

- Recall that ‘{}’ represents the empty set

Application: Set joins

- Recall queries of the form: “Find all pairs of documents (d_1, d_2) such that **some** | **not all** | **not only** | **no** | **all** | **only** words of d_1 are in d_2 .”
- These set-joins can be captured using the **overlap** and **containment** predicates
- To do so, we can define polymorphic user-defined functions that stand for these set-join predicates
- We will illustrate this for the **some** (i.e., **at least one**) and **all** set joins. The other set joins can be specified in a similar fashion

Application: Set joins

- **SOME** (at least one) set join

```
CREATE OR REPLACE FUNCTION atLeastOne (A anyarray, B anyarray)
RETURNS boolean AS
$$
SELECT A && B;
$$ LANGUAGE SQL;
```

- **ALL** set join (better called SUBSET join)

"Is **each** element in *A* an element of *B*?"

```
CREATE OR REPLACE FUNCTION Each (A anyarray, B anyarray)
RETURNS boolean AS
$$
SELECT A <@ B;
$$ LANGUAGE SQL;
```

Application: Set joins

We can then write queries with set joins as follows:

- “Find all pairs of documents (d_1, d_2) such that some words of d_1 are in d_2 .”

```
SELECT d1.doc, d2.doc  
FROM documents d1, documents d2  
WHERE atLeastOne(d1.words,d2.words)
```

- “Find all pairs of documents (d_1, d_2) such that all words of d_1 are in d_2 .”

Alternatively, “Find all pairs of documents (d_1, d_2) such that d_1 only contains words that are in d_2 .”

```
SELECT d1.doc, d2.doc  
FROM documents d1, documents d2  
WHERE Each(d1.words,d2.words)
```

Determining Set Size (Cardinality) ($|A|$)

- We may wish to determine the size (cardinality) of sets
- This can be done using the ARRAY **cardinality** function
- “Find the number of words in each document.”

```
SELECT d.doc, cardinality(d.words) AS number_of_words  
FROM documents d
```

doc	number_of_words
d1	3
d2	3
d3	2
d4	4
d5	2
d6	3
d7	3
d8	2

Example: Queries using set cardinality

- “Find the documents with fewer than 10 words”

```
SELECT d.doc
FROM documents d
WHERE cardinality(d.words) < 10
```

The UNNEST operator

- It is possible to **coerce** an array into a (unary) relation that contains the elements of the array
- This is done using the **UNNEST** operator

```
SELECT UNNEST(ARRAY[2,1,3,4,4]) → unnest
      2
      1
      3
      4
      4
```

- It is possible to provide an attribute name for the elements

```
SELECT UNNEST(ARRAY[2,1,3,4,4]) AS A → A
      2
      1
      3
      4
      4
```

Restructuring: the UNNEST operator

- It is possible to **restructure** a complex-object relation by using the **UNNEST** restructuring operator
- “Starting from the **documents** relation, create a relation of (doc, word) pairs.”

```
SELECT      d.doc, UNNEST(d.words) AS word  
FROM        documents d
```

doc	word
d1	A
d1	B
d1	C
d2	B
d2	C
d2	D
d3	A
d3	E
d4	B
d4	B
d4	A
d4	D
d5	E
d5	F
d6	A
d6	D
d6	G
d7	C
d7	B
d7	A
d8	B

Set operations: setUnion, setIntersection, and setDifference

- Using UNNEST and ARRAY construction it is also possible to define **setUnion**, **Intersection**, and **Difference** on sets represented as arrays
- We do this with polymorphic functions.
- Here we will show how to do this for **setUnion**

```
CREATE FUNCTION setUnion (A anyarray, B anyarray) RETURNS anyarray AS
$$
SELECT ARRAY( SELECT * FROM UNNEST(A)
UNION
SELECT * FROM UNNEST(B));
$$ LANGUAGE SQL;
```

```
SELECT setUnion( '{1, 2, 3}'::int[], '{2, 3, 3, 5}'::int[] );
```

```
setUnion
_____
{1, 2, 3, 5}
```

```
SELECT setUnion( '{"A", "B"}'::text[], '{"A", "C"}'::text[] );
```

```
setUnion
_____
{A, B, C}
```

Restructuring: GROUPING (nesting)

- Reconsider the `documentWord` relation
- “Restructure this relation by grouping the words of each document into a set (bag)“

`documentWord`

doc	word
d1	A
d1	B
d1	C
d2	B
d2	C
d2	D
d3	A
d3	E
d4	B
d4	B
d4	A
d4	D
d5	E
d5	F
d6	A
d6	D
d6	G
d7	C
d7	B
d7	A
d8	B

group words by doc
→

doc	words
d1	{A, B, C}
d2	{B, C, D}
d3	{A, E}
d4	{B, B, A, D}
d5	{E, F}
d6	{A, D, G}
d7	{C, B, A}
d8	{B, A}

Restructuring: GROUPING (nesting)

- This can be done using the **ARRAY** constructor operation

```
SELECT DISTINCT d.doc,  
    ARRAY(SELECT d1.word  
        FROM documentWord d1  
        WHERE d1.doc = d.doc) AS words  
    FROM documentWord d;
```

- Notice how the parameter **d** is used inside the **ARRAY** constructor to group together the words associated with the document **d**
- The **DISTINCT** operation is essential
- This query runs in $O(|documentWord|^2)$.

Restructuring: GROUPING (nesting) using the array_agg function

- The same restructuring can also be done using the **array_agg** aggregate function

```
SELECT d.doc, array_agg(d.word)
FROM documentWord d
GROUP BY (d.doc)
```

- The **GROUP BY(d.doc)** operation partitions the `documentWord` by `doc` values
- For each cell in this partition, the **array_agg** function aggregates in an array the words that are in that cell
- This query run in $O(|documentWord|)$
- So much faster than the other restructuring query

Repeated restructuring (Different views of same data)

- Starting from the **documents** relation, we may want to create a complex-object relation **words** which keeps for each word the set of documents that contain that word
- In other words, we want to do the following restructuring

doc	words
d1	{A, B, C}
d2	{B, C, D}
d3	{A, E}
d4	{B, B, A, D}
d5	{E, F}
d6	{A, D, G}
d7	{C, B, A}
d8	{B, A}

restructure:
Step 1: unnest on words;
Step 2: group docs by word
→

word	docs
A	{d1, d3, d4, d6, d7, d8}
B	{d1, d2, d4, d7, d8}
C	{d1, d2, d7}
D	{d2, d4, d6}
E	{d3, d5}
F	{d5}
G	{d6}

- This can be accomplished by unnesting the **documents** relation on words and then grouping the doc values by word

Repeated restructuring

```
WITH docWord AS (SELECT d.doc AS doc,  
                      UNNEST(d.words) AS word  
                 FROM documents d)  
SELECT p.word AS word, array_agg(p.doc) AS docs  
      FROM docWord p  
 GROUP BY (p.word)
```

Or, as one query

```
SELECT      word, array_agg(doc) AS docs  
FROM        (SELECT doc, UNNEST(words) AS word  
                  FROM documents d) p  
GROUP BY    (word)
```

Application: The word-count problem

“Determine the word-count, i.e., frequency of occurrence, of each word in the set of documents”

```
SELECT      word, cardinality(array_agg(doc)) AS wordCount
FROM        (SELECT doc, UNNEST(words) AS word
            FROM   documents d) p
GROUP BY    (word)
```

doc	words
d1	{A, B, C}
d2	{B, C, D}
d3	{A, E}
d4	{B, B, A, D}
d5	{E, F}
d6	{A, D, G}
d7	{C, B, A}
d8	{B, A}

→

word	wordCount
F	1
G	1
E	2
C	3
D	3
A	6
B	6

Application: The most frequent words

"Find the words that occur most frequently in the set of documents."

```
WITH E AS (
    SELECT word, cardinality(array_agg(doc)) AS wordCount
    FROM (SELECT doc, UNNEST(words) AS word
          FROM documents d) p
    GROUP BY (word))

SELECT word
FROM E
WHERE wordCount = (SELECT MAX(wordCount) FROM E)
```

Double nesting

- Consider the following [Enroll\(sid,cno,grade\)](#) relation

sid	cno	grade
1001	2001	A
1001	2002	A
1001	2003	B
1002	2001	B
1002	2003	A
1003	2004	A
1003	2005	B
1004	2002	A
1004	2004	A
1005	2001	B
1005	2003	A

- From this we want to create a complex-object relation which stores for each student, his or her courses, internally grouped by grades obtained in these courses
- This requires double nesting

Double nesting

- We begin by grouping on (sid,grade)

```
SELECT e.sid, e.grade, array_agg(e.cno) AS courses  
FROM enroll e  
GROUP BY (e.sid, e.grade)
```

- This gives the complex-object relation

sid	grade	courses
1001	A	{2001, 2002}
1001	B	{2003}
1002	A	{2003}
1002	B	{2001}
1003	A	{2004}
1003	B	{2005}
1004	A	{2002, 2004}
1005	A	{2003}
1005	B	{2001}

Double nesting

- We then group over the pair of attributes (grade,courses)

```
WITH F AS (SELECT e.sid, e.grade, array_agg(e.cno) AS courses  
        FROM enroll e  
        GROUP BY (e.sid, e.grade))
```

```
SELECT f.sid, array_agg((f.grade, f.courses)) AS grades  
      FROM F f  
      GROUP BY (f.sid)
```

- Notice the clause `array_agg((f.grade,f.course))`
- Recall that it is required to make a row `(f.grade,f.course)` since the `array_agg` function can only make an array wherein the array values are single values
- I.e., it is not allowed to write `array_agg(e.grade,e.course)`

Double nesting

sid	cno	grade
1001	2001	A
1001	2002	A
1001	2003	B
1002	2001	B
1002	2003	A
1003	2004	A
1003	2005	B
1004	2002	A
1004	2004	A
1005	2001	B
1005	2003	A

group by (cno)
group by(grade, courses)
→

sid	grades
1001	{"(A, {"2001, 2002"})", "(B, {"2003})"}
1002	{"(A, {"2003"})", "(B, {"2001})"}
1003	{"(A, {"2004"})", "(B, {"2005})"}
1004	{"(A, {"2002, 2004"})"}
1005	{"(A, {"2003})", "(B, {"2001})"}

- For example, student 1001 obtained two types of grades: 'A' and 'B'
- She received an 'A' in courses 2001 and 2002, and a 'B' is course 2003

Database Programming in an Object-Relational SQL Procedural Programming Language

PL/pgSQL - Procedural
Language/PostgreSQL

Dirk Van Gucht¹

¹Indiana University

March 30, 2019

Motivation

- Thus far, we have covered SQL and Object-Relational SQL (abbreviated as OR-SQL) as a language in which each statement corresponds to a **single** query, a **single** update, a **single** declaration (definition), etc.
- That means that a client application must send each query to the database server, wait for it to be processed, receive and process the results, do some computation, then send further queries to the server.
- All this incurs interprocess communication and will also incur network overhead if the client is on a different machine than the database server

Motivation

- In this lecture, we want to address writing **applications**, i.e., **programs**, wherein multiple OR-SQL can be bundled and processed using control statements such as assignment statements, conditional statements, loop statements, etc
- This correspond to writing programs in an imperative programming languages with the ability to use OR-SQL statements as **embedded code**
- We will use the PostgreSQL's `plpgsql` language to write such programs

Overview

- We begin with a general overview of the programming environment of `p1pgsql`
- We give a formal definition of a `p1pgsql` program and show how to run it in the PostgreSQL interpreter
- We will illustrate most of this using examples

plpgsql (**Declaration Statements**)

Type declarations

`CREATE TYPE`

Relation declarations

`CREATE TABLE`

View declarations

`CREATE VIEW`

Functions declarations

`CREATE FUNCTION`

Triggers declarations

`CREATE TRIGGER`

Program variable declarations

`DECLARE`

Iterator declaration

`FOR LOOP, FOREACH`

Cursor declaration

`DECLARE CURSOR`

plpgsql Garbage collection statements

Type	DROP TYPE
Relation	DROP TABLE
View	DROP VIEW
Function	DROP FUNCTION
Trigger	DROP TRIGGER
Cursor declaration	CLOSE CURSOR
Program variable declarations	not required
Iterator declarations	not required

Expressions and Statements

Expression	Any valid OR-SQL expression including SELECT FROM WHERE expression
Assignment statement	<code>variable_name := expression</code>
Assignment statement	<code>SELECT INTO variable_name</code>
Update statement	<code>INSERT, DELETE, UPDATE</code>
Return statement	<code>RETURN expression</code>
Return query statement	<code>RETURN QUERY query</code>
Function call	<code>SELECT function(parameters)</code>
Block statement	<code>BEGIN ... END</code>
Loop statement	<code>LOOP, WHILE, FOR</code>
Conditional statements	<code>IF ELSE, CASE</code>
Cursor operations	<code>OPEN, FETCH</code>

plpgsql (Program)

- The syntax of a plpgsql program is as follows:

```
CREATE OR REPLACE FUNCTION functionName (list of arguments)
    RETURNS return type AS
$$
<label>
<DECLARE declarations>
BEGIN
    sequence of statements;
END <label>;
$$ LANGUAGE plpgsql;
```

- A program is executed in the PostgreSQL interpreter using the call

```
SELECT functionName(parameters);
```

Program (Example with conditional if-else statement)

An example program with the **IF** statement

```
CREATE OR REPLACE FUNCTION convert(a char)
RETURNS float AS
$$
BEGIN
    IF (a = 't') THEN RETURN 1;
    ELSE
        IF (a = 'f') THEN RETURN 0;
        ELSE
            IF (a = 'u') THEN RETURN 0.5;
            ELSE RETURN(2);
            END IF;
        END IF;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

→

SELECT convert('u');	convert
<hr/>	
1/2	
<hr/>	
SELECT convert('z');	convert
<hr/>	
2	

Program (Example with conditional case statement)

An example program with the **CASE** statement

```
CREATE OR REPLACE FUNCTION convert(a char)
RETURNS float AS
$$
BEGIN
CASE WHEN (a = 't') THEN RETURN 1;
      WHEN (a = 'f') THEN RETURN 0;
      WHEN (a = 'u') THEN RETURN 0.5;
      ELSE RETURN 2;
END CASE;
END;
$$ LANGUAGE plpgsql;
```

→

SELECT convert('u');	convert
0.5	
SELECT convert('z');	convert
2	

Program (with loop statement)

Iterative program for the factorial(n) function

```
CREATE OR REPLACE FUNCTION factorial_Iterative (n integer)
RETURNS integer AS
$$
DECLARE
    result integer;
    i integer;
BEGIN
    result := 1;
    FOR i IN 1..n
        LOOP
            result := i * result;
        END LOOP;
    RETURN result;
END;
$$ language plpgsql;
```

Program (with recursion)

Recursive program for the factorial(n) function

```
CREATE OR REPLACE FUNCTION factorial_Recursive (n integer)
RETURNS integer AS
$$
BEGIN
    IF n = 0 THEN
        RETURN 1;
    ELSE
        RETURN n * factorial_Recursive(n-1);
    END IF;
END;
$$ language plpgsql;
```

Functions that affect the database state

- Functions can be defined to affect (change) the database state
- Often such functions do not need to return values: they have the **VOID** return type

```
CREATE OR REPLACE FUNCTION change_db_state()
    RETURNS VOID AS
$$
BEGIN
    DROP TABLE foo_relation;
    CREATE TABLE foo_relation(a integer);
    INSERT INTO foo_relation VALUES (1), (2), (3);
    DELETE FROM foo_relation WHERE a=1;
END;
$$ language plpgsql;
```

→

select change_db_state();
change_db_state
select * from foo_relation;
a
2
3

Program with local functions

- You can also CREATE local functions
- Care must be taken with function delimiters

```
CREATE OR REPLACE FUNCTION globalFunction()
RETURNS void AS
$proc$
BEGIN
    CREATE OR REPLACE FUNCTION localFunction()
    RETURNS integer AS
    $$
        SELECT 5;
    $$ language sql;
END;
$proc$ language plpgsql;
```

SELECT globalFunction();
globalfunction

→
SELECT localFunction();
localfunction

5

- Notice that `localFunction()` persists after the `SELECT globalFunction()` call

Two kinds of assignment statements

- The typical assignment statement is of the form
 $x := \text{expression};$
- An assignment to a **variable** can also be done with a query and the clause
 $\text{SELECT tuple component(s) INTO variable (s) FROM ... WHERE;}$
- The value of the tuple component (s) is (are) assigned to the **variable(s)**

```
CREATE OR REPLACE FUNCTION size_of_A()
    RETURNS integer AS
$$
DECLARE counter integer;
BEGIN
    SELECT INTO counter COUNT(*) from A;
    RETURN counter;
END;
$$ language plpgsql
```

→

SELECT * FROM A;
X
'A'
'B'
SELECT size_of_A();
size_of_A
2

Special alternative for SELECT INTO assignment statement

```
CREATE OR REPLACE FUNCTION size_of_A()
RETURNS integer AS
$$
DECLARE counter integer;
BEGIN
    SELECT INTO counter COUNT(*) from A;
    RETURN counter;
END;
$$ language plpgsql
```

Since the expression (SELECT COUNT(*) FROM A) evaluates to a single integer, this program can also be written as

```
CREATE OR REPLACE FUNCTION size_of_A()
RETURNS integer AS
$$
DECLARE counter integer;
BEGIN
    counter := (SELECT COUNT(*) from A);
    RETURN counter;
END;
$$ language plpgsql
```

SELECT INTO (non-deterministic behavior)

- SELECT INTO can lead to non-deterministic (random) effects!
- This is because SELECT INTO chooses the first available tuple from the result of the query and assigns it to the INTO variable (in our case the variable element_from_A).³
- Of course, this can be useful when sampling data

```
CREATE OR REPLACE FUNCTION choose_one_from_A()
RETURNS text AS
$$
DECLARE element_from_A text;
BEGIN
    SELECT INTO element_from_A x.a
    FROM (SELECT x from A ORDER BY random()) a;
    RETURN element_from_A;
END;
$$language plpgsql
```

→

<u>SELECT choose_one_from_A();</u>	<u>choose one from a</u>
	'B'
<u>SELECT choose_one_from_A();</u>	<u>choose one from a</u>
	'A'
<u>SELECT choose_one_from_A();</u>	<u>choose one from a</u>
	'A'

³If the query does not return any tuple, then the variable is set to NULL.

"Assignment" statements to relation variables

- “Assignment” statements to relation (table) variables are done using the **INSERT INTO**, **DELETE FROM**, and **UPDATE** statements, or using triggers

```
CREATE OR REPLACE FUNCTION relation_assignment()
RETURNS void AS
$$
BEGIN
CREATE TABLE IF NOT EXISTS AB(A integer, B integer);
DELETE FROM AB;
INSERT INTO AB VALUES (0,0);
INSERT INTO AB SELECT a1.x, a2.x FROM A a1, A a2;
UPDATE AB SET A = A*A WHERE B = 2;
END;
$$language plpgsql;
```

select * from A;

x
1
2

SELECT * FROM AB;
ERROR: relation "ab" does not exist

→

```
SELECT relation_assignment();
SELECT * from AB;
```

a	b
0	0
1	1
2	1
1	2
4	2

Iterators over collections

- Relations and arrays are **collections**
- Relations are unordered collections whereas arrays are ordered collections
- We consider **iterator variables** that slide (move; iterate) over such a collection **one element at a time**
- In SQL, an iterator variable over a relation (which may or may not be the result of a query) is often referred to as a **CURSOR**
- In SQL, it is frequently not necessary to use cursors as the following function illustrates

```
CREATE OR REPLACE FUNCTION there_is_book_that_cost_more_than(k integer)
RETURNS boolean AS
$$
SELECT EXISTS(SELECT * FROM book WHERE price > k);
$$ language sql
```

Iterators over collections (cursors)

```
CREATE OR REPLACE FUNCTION there_is_book_that_cost_more_than(k integer)
    RETURNS boolean AS
$$
BEGIN
    SELECT EXISTS(SELECT * FROM book WHERE price > k);
END;
$$ language sql;
```

The following function with the same semantics does use the iterator record variable (cursor) b

```
CREATE OR REPLACE FUNCTION there_is_book_that_cost_more_than(k integer)
    RETURNS boolean AS
$$
DECLARE exists_book boolean;
    b RECORD; – the structure will be defined during the program
BEGIN
    exists_book := false;
    FOR b IN SELECT * FROM book – RECORD b will have have the attribute structure of the book relation
    LOOP
        IF b.price > k
        THEN exists_book := true;
        EXIT;
        END IF;
    END LOOP;
    RETURN exists_book;
END; $$ language plpgsql;
```

Iterators over arrays

- Below is an example from the PostgreSQL manual illustrating iteration through an array using the **FOREACH** clause
- The function **sum** takes an integer array as input and returns the sum of its elements
- The variable **x** is the iterator which gets assigned, one at a time, to each **element** in the array
- Note in particular that **x** is **not assigned to index positions** of the array

```
CREATE FUNCTION sum(A int[])
  RETURNS int8 AS
$$DECLARE
  s int8 := 0;
  x int;
BEGIN
  FOREACH x IN ARRAY A
  LOOP
    s := s + x;
  END LOOP; RETURN s;
END;
$$ LANGUAGE plpgsql;
```

Iterators over arrays

On the right is an alternative version for the `sum` function.
There an index variable `i` is used that iterates over the index positions of the array.

```
CREATE FUNCTION sum(A int[])
RETURNS int8 AS
$$ DECLARE
  s int8 := 0;
  x int;
BEGIN
  foreach x IN ARRAY A
  LOOP
    s := s + x;
  END LOOP;
  RETURN s;
END;
$$ LANGUAGE plpgsql;
```

↔

```
CREATE FUNCTION sum(A int[])
RETURNS int8 AS
$$ DECLARE
  s int8 := 0;
  i int;
BEGIN
  FOR i IN array_lower(A,1)..array_length(A,1)
  LOOP
    s := s + A[i];
  END LOOP;
  RETURN s;
END;
$$ LANGUAGE plpgsql;
```

Key-value stores

**The MapReduce Data and Programming Model
(Simulated in Object-Relational SQL)**

**An Introduction to Distributed/Parallel Query Processing Based on
Data Partitioning**

Dirk Van Gucht¹

¹Indiana University

Key-values stores and queries

- A **key-value store** \mathbf{S} is a binary relation with **schema** (key: K , value: V) where K and V are **types** with **domains** $\text{dom}(K)$ and $\text{dom}(V)$ of **objects**
- Note that the key attribute K is not necessarily a primary key of \mathbf{S} . It is possible to have different key-value pairs in \mathbf{S} with the same key value
- A **key-value query** $q : \mathbf{S}_1 \rightarrow \mathbf{S}_2$ is a mapping that sends a key-value store $\mathbf{S}_1(K_1, V_2)$ to a key-value store $\mathbf{S}_2(K_2, V_2)$

Key-value query languages and data-compute environments

- A key-value query q can be programmed in any number of **(database) programming language** such as (Object-Relational) SQL, PHP, MapReduce, Hive, PigLatin, Scala, Python, Java, Javascript, R, scripting languages, etc
- The database programming language is implemented in a **data-compute environment (system)** which can centralized or distributed and which has a certain model of **data storage, data transfer, and communication**: MySQL, PostgreSQL, Hadoop, Spark, MongoDB, Cloudera Impala, Neo4J, Amazon AWS, etc

Key-value query languages and data-compute environments

- Database applications are often implemented and run in different such systems
- Performance can be greatly affected by this choice
- The **database-information ecosystem** is bewildering complex and varied, and keeps changing

The MapReduce Data and Programming Model

- In this lecture, we will introduce the **MapReduce programming language** as an early example of a **noSQL** language to program key-value pair queries
- MapReduce exhibits a variety of database modeling and query language concepts that have impacted newer database models and query languages

The MapReduce Data and Programming Model

- MapReduce as a programming environment is disappearing but is still very useful to discuss important concepts such as **key-value stores**, **data partitioning**, and **distributed query processing**
- We will cover the semantics of MapReduce and use the object-relational database model (such as implemented in PostgreSQL) to simulate it

The Word Count query (Running Example)

- A key-value store `Document(doc: text, words: text[])` stores a set of documents
- Each document is represented as a `(doc, words) pair`:
 - ① `doc` is the document identifier of the document
 - ② `words` is the set (bag) of words contained in the document

The Word Count query (Running Example)

- We wish to implement the word count query that maps the key-value store Document to the key-value store $\text{wordCount}(\text{word} : \text{text}, \text{wordcount} : \text{int})$
- The input to the word count query is the Document store
- The output of the word count query is the store of (word,wordCount) pairs where wordCount is the number of occurrences of the word word across all the documents in the Document store
- In summary, we want to implement

$\text{WordCount} : \text{Document}(\text{text}, \text{words}) \rightarrow \text{wordCount}(\text{word}, \text{wordcount})$

The Word Count query (Input)

- The store **Document** is created as follows:¹

```
CREATE TABLE Document(doc text, words text[]);
```

- Assume that we populate the **Document** store as follows:²

```
INSERT INTO Document VALUES ('d1', ARRAY['A','B','C']),  
('d2', ARRAY['B','C','D']),  
('d3', ARRAY['A','E']),  
('d4', ARRAY['B','B','A','D']),  
('d5', ARRAY['E','F']),
```

Document	
doc	words
d1	{A,B,C}
d2	{B,C,D}
d3	{A,E}
d4	{B,B,A,D}
d5	{E,F}

¹Notice that we represent a bag of words with an array.

²Notice that a word may occur multiple times in a document.

The Word Count query (Output)

Document

doc	words
d1	{A,B,C}
d2	{B,C,D}
d3	{A,E}
d4	{B,B,A,D}
d5	{E,F}

word count query



wordCount

word	wordcount
A	3
B	4
C	2
D	2
E	2
F	1

The Word Count query in object-relational SQL

- First formulation:

```
SELECT      p.word, CARDINALITY(ARRAY_AGG(p.doc)) AS wordCount
FROM        (SELECT d.doc, UNNEST(d.words) AS word
            FROM   Document d) p
GROUP BY    (p.word)
```

- Second formulation: Notice that we don't need the actual values of the document identifiers "doc". Thus, we can also formulate the word count query as follows:

```
SELECT      p.word, CARDINALITY(ARRAY_AGG(p.one)) AS wordCount
FROM        (SELECT UNNEST(d.words) AS word, 1 as one
            FROM   Document d) p
GROUP BY    (p.word)
```

The Word Count query in object-relational SQL (MapReduce style)

- Before we present the MapReduce simulation, we will decompose its object-relational SQL formulation into 3 phases that aligns with the MapReduce programming model
- This formulation will serve as a blueprint for the MapReduce simulation

WITH

%mapper phase:

```
doc_word AS (SELECT UNNEST(d.words) as word, 1 AS one
              FROM Document d),
```

%group (shuffle) phase:

```
word_ones AS (SELECT p.word, ARRAY_AGG(p.one) AS ones
                  FROM doc_word p
                  GROUP BY (p.word)),
```

%reducer phase:

```
word_count AS (SELECT q.word, CARDINALITY(q.ones) AS wordCount
                  FROM word_ones q)
```

%output:

```
SELECT word, wordCount FROM word_count
```

The Word Count query (mapper phase)

%mapper phase:

```
doc_word AS (SELECT UNNEST(d.words) as word, 1 AS one  
              FROM Document d),
```

Document	
doc	words
d1	{A,B,C}
d2	{B,C,D}
d3	{A,E}
d4	{B,B,A,D}
d5	{E,F}

mapper phase
→

doc_word	
word	one
A	1
B	1
C	1
B	1
C	1
D	1
A	1
E	1
B	1
B	1
A	1
D	1
E	1
F	1

The Word Count query (group (shuffle) phase)

%group (shuffle) phase:

```
word_ones AS (SELECT p.word, ARRAY_AGG(p.one) AS ones
               FROM doc_word p
               GROUP BY (p.word)),
```

doc_word	
word	one
A	1
B	1
C	1
B	1
C	1
D	1
A	1
E	1
B	1
B	1
A	1
D	1
E	1
F	1

group (shuffle) phase
→
group by word and aggregate the 1's

word_ones	
word	ones
A	{1,1,1}
B	{1,1,1,1}
C	{1,1}
D	{1,1}
E	{1,1}
F	{1}

The Word Count query (reducer phase)

%reducer phase:

```
word_count AS (SELECT q.word, CARDINALITY(q.ones) AS wordCount  
FROM word_ones q)
```

word_ones	
word	ones
A	{1,1,1}
B	{1,1,1,1}
C	{1,1}
D	{1,1}
E	{1,1}
F	{1}

reducer phase
→
per word, count its document occurrences

word_count	
word	wordCount
A	3
B	4
C	2
D	2
E	2
F	1

MapReduce queries (mapper and reducer functions)

- A basic MapReduce query is a pair of functions:
(mapper, reducer)
- The **mapper** function takes as input a *(key, value)*-pair and outputs a relation (bag) of *(key, value)*-pairs:

```
function mapper (key  $T_1$ , value  $T_2$ )
  returns table(key  $T_3$ , value  $T_4$ )
```

- The **reducer** function takes as input a *(key, bagOfValues)*-pair and outputs a relation (bag) of *(key, value)*-pairs:

```
function reducer (key  $T_3$ , values  $T_4[]$ )
  returns table(key  $T_5$ , value  $T_6$ )
```

- Notice how the output types T_3 and T_4 of the **mapper** function line up with the input types T_3 and $T_4[]$ of the **reducer** function

MapReduce queries

- A MapReduce query is a **sequence** (composition) of basic MapReduce queries
$$(\text{mapper}_1, \text{reducer}_1); (\text{mapper}_2, \text{reducer}_2); \dots; (\text{mapper}_n, \text{reducer}_n)$$
- The output type of the key-value pairs produced by reducer_i must be the same as the input type for mapper_{i+1} , for $i \in [1, n - 1]$

Semantics of a MapReduce query

- The semantics of a **basic** MapReduce query consists of a **mapper-**, a **group (shuffle-)**, and a **reducer-phase**:
 - In the **mapper phase**, the mapper is **map-applied**³ to an input relation of (key,value) pairs and the result is put into an intermediate relation *output_map* also consisting of (key,value) pairs
 - In the **group (shuffle) phase**, the *output_map* relation is grouped⁴ on its key attribute⁵, and for each key *k*, a pair (*k*, *bagOfValues(k)*) is produced, where
$$\text{bagOfValues}(k) = \{\{v \mid \text{output_map}(k, v)\}\},$$
i.e, the bag of all values in *output_map* with key value *k*
 - In the **reducer phase**, the reducer is **map-applied**⁶ to the (*key*, *bagOfValues*) pairs produced in the group (shuffle) phase. For each such pair, the reducer produces a relation of (*key*, *value*) pairs.
 - The **output** of the query is the union of all the relations produced in the **reduce_phase**
- The semantics of a MapReduce query is the **composition** of its basic MapReduce queries

³Typically in a parallelized and/or distributed manner.

⁴Sometimes also called *shuffled*

⁵Typically by hashing on this key-value.

⁶Typically in a parallelized and/or distributed manner.

MapReduce Limitations

- **Limitation 1:**
 - Notice that a MapReduce query does not have conditional statements nor loop statements
 - This is a serious limitation for problems that require iteration to find solutions
 - Such is the case in many data science and machine learning problems (for example page-rank algorithm, k-means clustering, gradient-descent in deep learning etc.)
- **Limitation 2:** Performance can be greatly affected by **skew in data**: during the grouping, different keys may be associated with vastly different numbers of values

MapReduce Limitations

- **Limitation 3:**
 - Between successive basic MapReduce query, intermediate results are written to files
 - This limits in-memory processing
- **Limitation 4:**
 - The input to a MapReduce program is a **single** relation.
 - Notice that therefore to do a **binary** operation such a join, union, intersection, or set-difference between two relations R and S , it is necessary store/model the data in both R and S into a **single** relation of key-values pairs— **awkward data modeling**

Simulating a (basic) MapReduce query in SQL

- At the beginning of the lecture, we formulated the **word count** query in object-relational SQL.
- We deliberately wrote it in a fashion that resembles the mapper-, group (shuffle)-, and reducer phases present in the semantics of a basic MapReduce query
- We can give an even more faithful simulation if we write this SQL query using a **mapper** function and a **reducer** function
- These functions can be programmed as Object-Relational SQL user-defined functions

The **mapper** function

- We formulate the **mapper** function as follows:

```
CREATE FUNCTION mapper(doc text, words text[])
    RETURNS TABLE (word text, one int) AS
$$
    SELECT UNNEST(words) as word, 1 as one;
$$ LANGUAGE SQL;
```

- The **mapper** function does the following when applied to a document (doc,words) pair:

```
SELECT word, one
FROM mapper('d1',ARRAY['A','A','B']);
```

word	one
A	1
A	1
B	1

The mapper function (map-apply simulation)

- The **mapper** function, when map-applied to the **Document** relation, produces the relation `map_output(word, one)`.

```
WITH map_output AS
  (SELECT q.word, q.one
   FROM Document d,
        LATERAL(SELECT p.word, p.one
                FROM  mapper(d.doc,d.words) p) q)
```

- Notice how we use the **LATERAL** clause
- This is convenient since we need to map-apply the **mapper** function to each document **d** in the **Document** store

The **mapper** function (map-apply)

Document

doc	words
d1	{A,B,C}
d2	{B,C,D}
d3	{A,E}
d4	{B,B,A,D}
d5	{E,F}

map-apply the **mapper** function



map_output

word	one
A	1
B	1
C	1
B	1
C	1
D	1
A	1
E	1
B	1
B	1
A	1
D	1
E	1
F	1

Simulation of group (shuffle) phase

- Before we specify the **reducer** function, we show how the **group-phase** is simulated
- This will be done by taking the map_output relation and grouping it on the key attribute **word**

```
WITH group_output AS
  (SELECT p.word, ARRAY_AGG(p.one) as ones
   FROM map_output p
   GROUP BY (p.word))
```

Simulation of the group (shuffle) phase

map_output

word	one
A	1
B	1
C	1
B	1
C	1
D	1
A	1
E	1
B	1
B	1
A	1
D	1
E	1
F	1

group (shuffle)
→

group_output

word	ones
A	{1,1,1}
B	{1,1,1,1}
C	{1,1}
D	{1,1}
E	{1,1}
F	{1}

The **reducer** function

- We now formulate the **reducer** function.
- In our case, this function takes as input a
 $(\text{word}, \{\!\!\{1, \dots, 1\}\!\!\})$
pair and outputs the desired
 $(\text{word}, \text{COUNT}(\{\!\!\{1, \dots, 1\}\!\!\}))$
pair (in a relation)

```
CREATE FUNCTION reducer(word text, ones int[])
    RETURNS TABLE(word text, wordCount int) AS
$$
    SELECT word, CARDINALITY(ones) as wordCount;
$$ LANGUAGE SQL;
```

- The **reducer** function does the following when applied to a (word,ones) pair:

```
SELECT word, wordCount
FROM    reducer('A','\{1,1,1,1\}');
```

word	wordCount
A	4

The **reducer** function (map-apply simulation)

- We can now show the simulation of the **reducer** phase:
- I.e., map-apply the **reducer** function to the (word,ones) pairs generated in the **group (shuffle) phase** and produce the output of the word count MapReduce query

```
SELECT r.word, r.wordCount
  FROM group_output q,
       LATERAL(SELECT p.word, p.wordCount
              FROM reducer(q.word, r.ones) p) r
```

- So we have

group_output	
word	ones
A	{1,1,1}
B	{1,1,1,1}
C	{1,1}
D	{1,1}
E	{1,1}
F	{1}

map-apply the **reducer** function
→

word	wordCount
A	3
B	4
C	2
D	2
E	2
F	1

The word count MapReduce query in Object-Relational SQL

```
CREATE FUNCTION mapper(doc text, words text[])
    RETURNS TABLE (word text, one int) AS
$$
    SELECT UNNEST(words), 1
$$ LANGUAGE SQL;
```

```
CREATE FUNCTION reducer(word text, ones int[])
    RETURNS TABLE(word text, wordCount int) AS
$$
    SELECT word, CARDINALITY(ones);
$$ LANGUAGE SQL;
```

MapReduce simulation in Object-Relational SQL

- Putting everything together, we get the following simulation of the **word count** MapReduce query in Object-Relational SQL:

```
WITH
    %mapper phase
    map_output AS
        (SELECT q.word, q.one
         FROM Document d,
              LATERAL(SELECT p.word, p.one
                     FROM mapper(d.doc,d.words) p) q),
    %group phase
    group_output AS
        (SELECT p.word, array_agg(p.one) as ones
         FROM map_output p
         GROUP BY (p.word)),
    %reducer phase
    reduce_output AS
        (SELECT r.word, r.wordCount
         FROM group_output q,
              LATERAL(SELECT p.word, p.wordCount
                     FROM reducer(q.word, q.ones) p) r)
    %output
    SELECT word, wordCount
    FROM reduce_output;
```

MapReduce in distributed setting

- In a distributed setting of compute nodes connected by a network, the Document key-value store is stored or can be partitioned into chunks that reside at or can be distributed to the local file systems of the compute nodes
- There are multiple copies of the same mapper function at the compute nodes these can be evaluated (i.e., map-applied) independently and in parallel (there is no shared memory nor explicit communication)
- More specifically, an instance of the mapper function processes the chunk of Document at its compute node and provide its output to the group (shuffle) process
- The group (shuffle) phase is typically implemented by applying a hash-function to the keys of the (key,value) pairs emitted by the mappers. Applied to a key, this hash-function will give the location of another compute node
- A (key,value) pair is sent to the compute node with that key's hash-function value. Notice that, because of the properties of hashing functions, all (key,value) pairs with the same key will be sent to the same compute node

MapReduce in distributed setting

- There are multiple copies of the **reducer** function waiting at the compute nodes where the hashed (key,value) pairs arrive
- After **all** the appropriate values for a key have been sent and received by the appropriate compute nodes, the reducers can go to work locally (at the compute node) on the list of values associated with a key
- The reducers can transmit their output, or they can keep it locally for further processing by other MapReduce queries.
- A big problem is skew in the data. It is possible that there is an uneven distribution of the values associated with keys. In that case, the computation can be slowed at the reducer compute nodes and the benefits of parallel (distributed) computing can be lost.

The word count MapReduce query in Object-Relational SQL (Chunks/Partitioned Data)

- Assume that the document database Document is partitioned into a collection of pairwise disjoint document databases $\text{Document}_1, \dots, \text{Document}_n$:

$$\text{Document} = \text{Document}_1 \cup \dots \cup \text{Document}_n.$$

- We can use the same **mapper** function and apply it independently and in parallel to each of the Document_i .
- The results of these n **mapper** computations can be accumulated and then grouped (shuffled) and sent to the **reducer** function.
- Observe that the **mapper** and **reducer** functions do not need to be changed.
- The simulation of this process is shown on the next slide

The word count MapReduce query in Object-Relational SQL (Alternative; Partitioned Data)

```
WITH
%mapper phase— mapper function is evaluated on separate document chunks and accumulated with UNION
map_output AS
  (SELECT q.word, q.one
   FROM Document1 d1,
        LATERAL(SELECT p.word, p.one
                 FROM mapper(d1.doc,d1.words) p)
   UNION
   ...
   UNION
   SELECT q.word, q.one
   FROM Documentn dn,
        LATERAL(SELECT p.word, p.one
                 FROM mapper(dn.doc,dn.words) p) q)
%group phase
group_output AS
  (SELECT p.word, array_agg(p.one) as ones
   FROM map_output p
   GROUP BY (p.word)),
%reducer phase
reduce_output AS
  (SELECT r.word, r.wordCount
   FROM group_output q,
        LATERAL(SELECT p.word, p.wordCount
                 FROM reducer(q.word, q.ones) p) r)
%output
SELECT word, wordCount
FROM reduce_output;
```

The word count MapReduce query in Object-Relational SQL (Alternative; local word counts)

- In the MapReduce query below, we empower the **mapper** to do a local word count problem **within** a single document
- The reducer will then receive, for each word, the local word counts for that word in each document
- The sum of these local word counts is the total word count.

```
CREATE FUNCTION mapper(doc text, words text[])
    RETURNS TABLE (word text, localWordCount bigint) AS
$$
    SELECT w.word, COUNT(1)
    FROM  (SELECT UNNEST(words) AS word) w
    GROUP BY (w.word)
$$ LANGUAGE SQL;

CREATE FUNCTION reducer(word text, localWordCounts int[])
    RETURNS TABLE(word text, wordCount bigint) AS
$$
    SELECT reducer.word, SUM(p.localWordcount)
    FROM (SELECT UNNEST(localWordCounts) AS localWordcount) p
$$ LANGUAGE SQL;
```

- This query does more local computation but requires less data transfer during the group/shuffle phase

The word count MapReduce query in Object-Relational SQL (Alternative; different input pairs)

- In the MapReduce query below, we assume that the input is a `docWord` key-value store of type (doc: text, word:text)
- A pair (d, w) is in `docWord` if word w occurs in document d
- So we need to write a different mapper
- The reducer will not need to be changed

```
CREATE FUNCTION mapper(doc text, word text)
    RETURNS TABLE (word text, one int) AS
$$
    SELECT word, 1 as one
$$ LANGUAGE SQL;

CREATE FUNCTION reducer(word text, localWordCounts int[])
    RETURNS TABLE(word text, wordCount bigint) AS
$$
    SELECT reducer.word, SUM(p.localWordcount)
        FROM (SELECT UNNEST(localWordCounts) AS localWordcount) p
$$ LANGUAGE SQL;
```

The word count MapReduce query in Object-Relational SQL (Extracting words in sentences)

- In the MapReduce query below, we assume that the input is a **Sentences** key-value store of type (sid: int, sentence:text)
- A pair (n, s) is in **Sentences** if n is the id of a sentence s
- So we need a different mapper to process such a pair⁷

```
CREATE FUNCTION mapper(sid int, sentence text)
    RETURNS TABLE (word text, one int) AS
$$
    SELECT UNNEST(regexp_split_to_array(sentence, '\s+')) as word, 1 as one
$$ LANGUAGE SQL;

CREATE FUNCTION reducer(word text, localWordCounts int[])
    RETURNS TABLE(word text, wordCount bigint) AS
$$
    SELECT reducer.word, SUM(p.localWordcount)
        FROM (SELECT UNNEST(localWordCounts) AS localWordcount) p
$$ LANGUAGE SQL;
```

- The function `regexp_split_to_array(sentence, '\s+')` is a PostgreSQL function that takes a sentence and creates an array of the words in that sentence

⁷Observe that the reducer does not need to be changed.

Key-value stores:
The Spark Data and Programming Model
**(Explained relative to Relational Algebra
and Object-Relational SQL)**

**An Introduction to Distributed/Parallel
Query Processing Based on Data
Partitioning**

Dirk Van Gucht¹

Key-values stores and queries

- A key-value store \mathbf{S} is a relation with schema
(key: K , value: V)
where K and V are types with domains $\text{dom}(K)$ and
 $\text{dom}(V)$ of objects
- A key-value pair (k, v) in $\mathbf{S}(K, V)$ is an element of
 $\text{dom}(K) \times \text{dom}(V)$
- A key-value query $q : \mathbf{S}_1 \rightarrow \mathbf{S}_2$ is a mapping that sends a
key-value store $\mathbf{S}_1(K_1, V_1)$ to a key-value store $\mathbf{S}_2(K_2, V_2)$

The Spark data and programming model (Data Model)

- Resilient Distributed Datasets (RDDs): collection of elements that can be operated on in parallel
- Elements can be any type. Typically, however, they are key-value pairs.
- There are two ways to further create RDDs
 - 1 parallelizing an existing (RDD) collection

```
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data)
```
 - 2 referencing a dataset in an external storage system (not further discussed)

Spark data and programming model (Data Model)

- Spark permits the definition of functions to create (key, value) pairs.
- In many ways, just as MapReduce, Spark processes key-value stores
- Below is an example of (key-value) creation in Spark:

Input RDD
String
hello
world
how
are
you

```
.map(word => Tuple2(word, 1))  
      .map(word => (word, 1))
```

Output RDD	
String	
Int	
hello	1
world	1
how	1
are	1
you	1

Spark programming model

- Spark supports two types of operations on RDDs:
 - 1 **transformations**, which create a new RDD dataset from an existing RDD
 - 2 **actions**, which return a value to the driver program after running a computation on the dataset
- .map(word => (word,1)).reduceByKey(lambda a,b: a+b)
- Transformations and actions are written as functions that use **algebraic operations** most of which correspond directly to operations in Relational Algebra (join, selection, union, etc) and Object-Relational SQL (GROUP BY, aggregate functions, and UNNEST)

Spark programming model (related database concepts)

- All transformations in Spark are **lazy**: they do not compute their results immediately:
database concept: [views](#)
- The transformations are only computed when an action requires a result:
database concept:
[query](#) evaluation on data represented by [views](#)
- The Spark programming model permits compilation and optimization:
database concept:
[query translation](#) and [query optimization](#)

Spark programming model (Persistent RDDs)

- By **default**, each transformed RDD is recomputed each time you run an action on it
database concept: just like **views** are lazily evaluated
- However, you may also **persist** a (transformed) RDD in memory using the **persist** (or **cache**) method
- Spark will keep the elements around on the cluster for much faster access the next time you query it
database concept: just like **materialized views**

Spark programming model (Transformations)

Spark	SQL/RA
<code>R.map(func)</code>	<code>SELECT func(r) FROM R r</code>
<code>(R₁, ..., R_n).mapPartitions(func)</code>	<code>SELECT func(r₁) FROM R₁ r₁</code> <code>UNION ... UNION</code> <code>SELECT func(r_n) FROM R_n r_n</code>
<code>R.filter(func)</code>	<code>SELECT r.* FROM R r</code> <code>WHERE func(r)</code>
<code>R.flatMap(func)</code>	<code>SELECT UNNEST(func(r))</code> <code>FROM R r</code>
<code>R.union(S)</code>	<code>SELECT r FROM R r</code> <code>UNION</code> <code>SELECT s FROM S s</code>
<code>R.intersection(S)</code>	<code>SELECT r FROM R r</code> <code>INTERSECT</code> <code>SELECT s FROM S s</code>

Spark programming model (Transformations)

Spark	SQL/RA
<code>R.distinct()</code>	SELECT DISTINCT r.* FROM R r
<code>Rk,v.groupByKey()</code>	SELECT K, array_agg(V) FROM Rk,v GROUP BY(K)
<code>Rk,v.reduceByKey(func)</code>	SELECT K, func(array_agg(V)) FROM Rk,v GROUP BY(K)
<code>Rk,v.sortByKey()</code>	SELECT r.* FROM Rk,v r ORDER BY(K)
<code>Rk,v.join(Sk,w)</code>	SELECT r.K,(r.V,s.W) FROM R r NATURAL JOIN S s
<code>R.cartesian(S)</code>	SELECT (r.*,s.*) FROM R r CROSS JOIN S s

Spark programming model (The co-group transformation)

$R_{K,V}.cogroup(S_{K,W})$

R		S		cogroup →	
K	V	K	W		
a	1	a	1		
a.	2	a	3		
b.	1	c	2		
c.	3	d	1		
		d	4		

K	(RV_values, SW_values)
a	({1,2}, {1,3})
b	({1}, {})
c	({3}, {2})
d	({}, {1,4})

```
WITH Kvalues AS (SELECT r.K FROM R r UNION SELECT s.K FROM S s),
      R_K AS (SELECT k.K,
                      ARRAY(SELECT r.V
                            FROM R r WHERE r.K = k.K) AS RV_values
                     FROM Kvalues k),
      S_K AS (SELECT k.K,
                      ARRAY(SELECT s.W
                            FROM S s WHERE s.K = k.K) AS SW_values
                     FROM Kvalues k)

      SELECT K, (RV_values, SW_values)
      FROM R_K NATURAL JOIN S_K
```

Observe that $R \bowtie S$ can be derived by applying flattening operations.

SQL to RA to Spark

- Translate SQL query to RA expression
- Optimize RA expression
- Translate RA expressions on the basis of the following correspondences:

σ_C	\rightarrow	.filter(func _C)
π_C	\rightarrow	.map(func _L)
\times	\rightarrow	.cartesian
\bowtie	\rightarrow	.join
U	\rightarrow	.union
\cap	\rightarrow	.intersection
$-$	\rightarrow	.except or minus

Object-relational SQL queries with aggregate functions can be similarly translated using .groupByKey, reduceByKey(func)

Spark (Actions)

A Spark action triggers the evaluation of a program (including evaluation of the transformations)

.reduce(func)	Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
.collect()	Return all the elements of the dataset as an array at the driver program.
.count()	Return the number of elements in the dataset.
.first()	Return the first element of the dataset (similar to take(1)).
.take(n)	Return an array with the first n elements of the dataset.
.countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.

Complications during distributed computation due to partitioned data

RDDs R and S are stored as partitions so that

$$R = R_1 \cup \dots \cup R_m$$

The unary operations π and σ can be efficiently implemented in a parallel/distributed system

$$\pi_L(R) = \pi_L(R_1) \cup \dots \cup \pi_L(R_m)$$

$$\sigma_C(R) = \sigma_C(R_1) \cup \dots \cup \sigma_C(R_m)$$

Complications during distributed computation due to partitioned data

$$\begin{array}{lcl} R & = & R_1 \cup \dots \cup R_m \\ S & = & S_1 \cup \dots \cup S_n \end{array}$$

The binary operations \cup , \cap , $-$, \bowtie and \times may require extensive data communication and transfer:

$R[\cup \cap] S = \bigcup_{i,j} R_i [\cup \cap] S_j$	$R - S = \bigcup_{i,j} R_i - S_j$
$R \bowtie S = \bigcup_{i,j} R_i \bowtie S_j$	$R \times S = \bigcup_{i,j} R_i \times S_j$

- Notice that we get a **quadratic number $m \times n$** of operations to perform!
- Data needs to be **shuffled** which is expensive.
- These problems get only worse when there are many RDDs that are part of a query such as $(R \bowtie S) - T$.

Spark: Shared Variables

Spark does provide two limited types of shared variables:

- 1 **broadcast variables**: Broadcast variables allow the programmer to keep a **read-only variable** cached on each machine rather than shipping a copy of it with tasks
- 2 **accumulators**: Accumulators are variables that are only “added” to through an associative and commutative operation and can therefore be efficiently supported in parallel.
Compute nodes can add to the accumulator (but not see it). Only driver see accumulator.

Nested Relational and Semi-Structured Databases

Databases with JSON Objects

Dirk Van Gucht¹

¹Indiana University

Motivation

- In the relational database model, each relation has a schema of attributes with **atomic domains** such as booleans, numbers, text strings etc
- In **complex-object databases**, data is permitted to have objects with domains such as arrays, sets, bags, objects of composite types (rows), relations, **JSON** objects, etc.

Motivation

- For such databases, we need mechanisms to
 - ① **define** domains of complex-object types
 - ② **search (query)** and **manipulate** databases with complex objects
- We will focus on and contrast two kinds of databases that are recursively defined in terms of
 - ① complex object types: row (record) and set (bags);¹
 - ② the JSON type.²

¹Such database are called **nested relational databases**.

²Such database are called **semi-structured databases**.

Complex-object types and nested relations

- Complex object types are inductively defined starting from atomic types and then using the row type and set type constructors:
 - The types int, text, boolean, etc are **atomic types**
 - If A_1, \dots, A_k are distinct attribute name and τ_1, \dots, τ_k are atomic types or set types, then
$$(A_1 : \tau_1, \dots, A_k : \tau_k)$$
is a **composite type**³
 - If τ is a composite type then $\{\tau\}$ is a **set type**
- If R is a relation name and τ is a **composite** type, then $R : \tau$ is a **nested relation** of type τ ⁴

³A composite type is also called a record or row type.

⁴Sometimes also called a complex-object relation of type τ .

Complex-objects domain and nested relation instances

- Complex object domains are inductively defined starting from domains of atomic types:
 - Each atomic type τ has a domain $dom(\tau)$ of atomic values such as integers, text, boolean etc.
 - If τ is a composite type $(A_1 : \tau_1, \dots, A_k : \tau_k)$, then
$$dom(\tau) = dom(\tau_1) \times \dots \times dom(\tau_k)$$
wherein the i -th component of an object $o \in dom(\tau)$ can be identified as $o.A_i$ for each $i \in [1, k]$
 - If $\{\tau\}$ is a set type, then $dom(\{\tau\})$ is the set of all finite subsets of $dom(\tau)$. So each object $o \in dom(\{\tau\})$ is a finite set of tuples of type τ
- If $R : \tau$ is a nested relation, then a nested relation instance of type $R : \tau$ is a finite set of tuples in $dom(\tau)$.

Examples of complex-object types

The following are examples of complex object types:

int	atomic type
text	atomic type
boolean	atomic type
(sid: int, sname: text, birthYear: int)	composite type
{(word: text)}	set type
(sid: int, gradeInfo: {(grade: text, courses: {(cno: int)})})	composite type

For succinctness, we will often omit the atomic types:

$(sid, sname, birthYear)$	composite type
$\{(word)\}$	set type
$(sid, gradeInfo \{(grade, courses \{(cno)\})\})$	composite type

Examples of nested relations

The following are examples of types of nested relations:

Student: (sid, sname, birthYear)

documentWords: (doc, words {(word)})

studentGrades: (sid, gradeInfo {(grade, courses {(cno)}}})

Course: (cno, name, students {(sid, name, majors {(major)}}), teachers {(tid, name)}})

Defining complex-object types in SQL

In (object-relational) SQL,

- Composite types are defined using **composite type declaration**:

```
CREATE TYPE studentType AS (sid int, sname text, birthYear text);
```

```
CREATE TYPE courseType AS (cno int);
```

```
CREATE TYPE studentType AS (sid int);
```

- Set types are defined using the **array type constructor**

```
CREATE TYPE gradeCoursesType AS (grade text, courses courseType[]);
```

```
CREATE TYPE gradeStudentsType AS (grade text, students studentType[]);
```

Defining nested relations in SQL

```
CREATE TYPE gradeCoursesType AS (grade text, courses courseType[ ]);
```

```
CREATE TYPE gradeStudentsType AS (grade text, students studentType[ ]);
```

In SQL, **nested relations** are defined using the **CREATE TABLE** statement

```
CREATE TABLE student(sid int, sname text, birthYear text);
```

```
CREATE TABLE course(cno int, cname text, dept text);
```

```
CREATE TABLE enroll(sid int, cno int, grade text);
```

```
CREATE TABLE studentGrades(sid int, gradeInfo gradeCoursesType[]);
```

```
CREATE TABLE courseGrades(cno int, gradeInfo gradeStudentsType[]);
```

Specifying nested relation instances using INSERT INTO (Example)

```
CREATE TYPE fooType AS (C int, D int);
CREATE TYPE barType AS (A int, B fooType[]);

CREATE TABLE tableTest (X int, Y barType[]);
```

```
insert into tableTest values (1, ARRAY[]::barType[]);
insert into tableTest values (2, ARRAY[(2,ARRAY[(3,4)]::fooType[])])::barType[]);
insert into tableTest values (3, ARRAY[(3,ARRAY[(4,5)]::fooType[]),(4,ARRAY[(5,6),(6,7)]::fooType[])])::barType[]);
```

We obtain the following nested relation instance:

tableTest

X	Y
1	{}
2	{(2,{(3,4)})}
3	{(3,{(4,5)}), (4,{(5,6), (6,7)})}}

Nested relations in SQL

A better visualization of the nested relation instance is as follows:

testTable	
X	Y
1	{}
2	{(2,{(3,4)})}
3	{(3,{(4,5)}), (4,{(5,6), (6,7)})}

nested relation visualization



testTable									
X	Y								
1	<table border="1"><tr><td>A</td><td>B</td></tr><tr><td></td><td></td></tr></table>	A	B						
A	B								
2	<table border="1"><tr><td>A</td><td>B</td></tr><tr><td>2</td><td><table border="1"><tr><td>C</td><td>D</td></tr><tr><td>3</td><td>4</td></tr></table></td></tr></table>	A	B	2	<table border="1"><tr><td>C</td><td>D</td></tr><tr><td>3</td><td>4</td></tr></table>	C	D	3	4
A	B								
2	<table border="1"><tr><td>C</td><td>D</td></tr><tr><td>3</td><td>4</td></tr></table>	C	D	3	4				
C	D								
3	4								
3	<table border="1"><tr><td>A</td><td>B</td></tr><tr><td>3</td><td><table border="1"><tr><td>C</td><td>D</td></tr><tr><td>4</td><td>5</td></tr></table></td></tr></table>	A	B	3	<table border="1"><tr><td>C</td><td>D</td></tr><tr><td>4</td><td>5</td></tr></table>	C	D	4	5
A	B								
3	<table border="1"><tr><td>C</td><td>D</td></tr><tr><td>4</td><td>5</td></tr></table>	C	D	4	5				
C	D								
4	5								
4	<table border="1"><tr><td></td><td></td></tr><tr><td>C</td><td>D</td></tr><tr><td>5</td><td>6</td></tr><tr><td>6</td><td>7</td></tr></table>			C	D	5	6	6	7
C	D								
5	6								
6	7								

Nested Relation instance: JSON representation

An alternative specification of the example nested relation instance is as the following JSON object:

```
[{"X": 1,
 "Y": [],
 {"X": 2,
 "Y": [{"A": 2,
          "B": [{"C": 3,
                  "D": 4}]}]},
 {"X": 3,
 "Y": [{"A": 3,
          "B": [{"C": 3,
                  "D": 4}]}},
 {"A": 4,
 "B": [{"C": 5,
          "D": 6},
 {"C": 6,
 "D": 7}]}}]
```

Example: Populating a nested relation instance with an INSERT INTO Query

- We assume that data has been inserted into the Enroll relation
- We can then populate the `studentGrades` nested relation with an INSERT INTO statement using a query with repeated grouping (nesting) and array aggregation:

```
INSERT INTO studentGrades
WITH E AS
  (SELECT sid, grade, array_agg(row(cno)::courseType) AS courses
   FROM Enroll
   GROUP BY (sid, grade)),
F AS
  (SELECT sid, array_agg(row(grade, courses)::gradeCoursesType) AS gradeInfo
   FROM E
   GROUP BY(sid));
SELECT sid, gradeInfo
FROM F
```

Example: Populating a nested relation with an INSERT INTO Query

We have achieved the following:

Enroll

sid	cno	grade
100	200	A
100	201	B
100	202	A
101	200	B
101	201	A
102	200	B
103	201	A
101	202	A
101	301	C
101	302	A
102	202	A
102	301	B
102	302	A
104	201	D

double grouping:
group by (sid,grade)
group by (sid)
→
double nesting

studentGrades

sid	gradeInfo
100	{(A,{(200),(202)}),(B,{(201)})}
101	{(B,{(200)}),(A,{(201),(202),(302)}),(C,{(301)})}
102	{(A,{(202),(302)}),(B,{(200),(301)})}
103	{(A,{(201)})}
104	{(D,{(201)})}

- For example, student 100 obtained two types of grades: 'A' and 'B'
- She received an 'A' in courses 200 and 202, and a 'B' in course 201

Querying nested relations

- Just as in the case for ordinary relations, nested relations can be queried in SQL using its standard query constructs
- Because of the **hierarchical** structure of such relations, most queries need to navigate in accordance with this hierarchy
- Frequently, the complex-objects of set-type need to be **unnested** to reveal their inner data
- In addition, and depending on the output structure of the query, **grouping and array (set) aggregation** (nesting) needs to be applied

Example: Querying nested relations

The following query illustrates **path navigation** along a path in the nested relation hierarchy

- “Find the grade information of each student who received an ‘A’ in some course.”

```
SELECT sid, gradeInfo  
FROM studentGrades sg  
WHERE 'A' IN (SELECT grade  
               FROM UNNEST(sg.gradeInfo))
```

- Notice the **UNNEST(sg.gradeInfo)** operation
- This UNNEST operation permits us to access the tuples that reside in the sg.gradeInfo set object.
- Thus, in particular, we can access the grade attribute of these tuples and compare their grade components with the grade ‘A’

Example: Querying nested relations

- “Find the grade information of each student who received an ‘A’ in some course.”

```
SELECT sid, gradeInfo  
FROM studentGrades sg  
WHERE 'A' IN (SELECT grade  
              FROM UNNEST(sg.gradeInfo))
```

- We obtain the following result:

sid	gradeInfo
100	{(A,{(200),(202)}),(B,{(201)})}
101	{(B,{(200)}),(A,{(201),(202),(302)}),(C,{(301)})}
102	{(A,{(202),(302)}),(B,{(200),(301)})}
103	{(A,{(201)})}

Example: Querying nested relations

- “Find the grade information of each student who received an ‘A’ in some course.”
- We can express this query also as follows:

```
SELECT sid, gradeInfo  
FROM studentGrades sg, UNNEST(sg.gradeInfo) g  
WHERE g.grade = 'A'
```

Example: Querying nested relations

The following query illustrates extraction of sub-objects inside other objects:

- “Find for each student the set of courses in which he or she received an ‘A’ ”

```
SELECT sg.sid, g.courses
FROM studentGrades sg, UNNEST(sg.gradeInfo) g
WHERE g.grade = 'A'
```

- We obtain the following result:

sid	courses
100	{(200),(202)}
101	{(201),(202),(302)}
102	{(202),(302)}
103	{(201)}

Example: Querying nested relations

The following query illustrates "path" navigation along a path in the nested relation hierarchy

- “Find the grade information of each student who enrolled in course 301.”

```
SELECT sid, gradeInfo  
FROM studentGrades sg,  
      UNNEST(sg.gradeInfo) g,  
      UNNEST(g.courses) c  
WHERE c.cno = 301;
```

- In this case we need to UNNEST twice since the cno information resides at the 2nd level in the hierarchy
- Notice how this query follows the hierarchical structure of the **studentGrades** relation

Example: Querying nested relations

The following query illustrates "path" navigation along a path in the nested relation hierarchy where, in addition, conditions are checked at nodes in the path:

- "Find the grade information of each student who received a 'B' in course 301."

```
SELECT    sid, gradeInfo  
FROM      studentGrades sg,  
          UNNEST(sg.gradeInfo) g,  
          UNNEST(g.courses) c  
WHERE     g.grade = 'B' AND c.cno = 301;
```

I.e, in studentGrades, navigate along each path with the following structure and conditions:

$$sg \rightarrow g[grade = 'B'] \rightarrow c[cno = 301]$$

In Xpath (Unix-like) path notation:

$$sg/g[grade = 'B']/c[cno = 301]$$

Example: Querying nested relations

The following query illustrates **path navigation** followed by **object construction**:

- “For each student, find the set of courses in which he or she is enrolled.”

```
SELECT      sid, array_agg(c.cno) as courses
FROM        studentGrades sg,
            UNNEST(sg.gradeInfo) g,
            UNNEST(g.courses) c
GROUP BY    (sg.sid)
```

- The result is as follows:

sid	courses
100	{200,201,202}
101	{200,201,202,301,302}
102	{200,202,301,302}
103	{201}
104	{201}

Example: Querying nested relations

The following query illustrate **path navigation**, **joining** nested relations, **object construction**, and **conditions** checking:

“For each student who majors in ‘CS’, list his or her sid and sname, along with the courses she is enrolled in. Furthermore, these courses should be grouped by the department in which they are offered.”

sid	sname	courseInfo
101	Nick	$\{(CS,\{(301, AI), (200, PL), (202, Dbs)\}), (Math,\{(201, Calculus)\}), (Philosophy,\{(302, Logic)\})\}$
102	Chris	$\{(CS,\{(200, PL), (202, Dbs), (301, AI)\}), (Philosophy,\{(302, Logic)\})\}$
103	Dinska	$\{(Math,\{(201, Calculus)\})\}$
105	Vince	$\{\}$

Example: Nested relations restructuring

“For each student who majors in ‘CS’, list her sid and sname, along with the courses she is enrolled in. Furthermore, these courses should be grouped by the department in which they are offered.”

```
WITH E AS (SELECT sid, cno
            FROM   studentGrades sg,
                   unnest(sg.gradeInfo) g,
                   unnest(g.courses) sc),

      F AS (SELECT sid, dept, array_agg((cno,cname)) as courses
            FROM   E NATURAL JOIN Course
            GROUP  BY(sid, dept))

SELECT sid, sname, ARRAY(SELECT (dept, courses)
                         FROM   F
                         WHERE   s.sid = F.sid) AS courseInfo
FROM   student s
WHERE  sid IN (SELECT sid
                  FROM   major m
                  WHERE  major = 'CS');
```

Example: Nested relations restructuring

“For each student who majors in ‘CS’, list her sid and sname, along with the courses she is enrolled in. Furthermore, these courses should by grouped by the department in which they are offered.”

We obtain the following result:

sid	sname	courseInfo
101	Nick	$\{(CS,\{(301, AI), (200, PL), (202, Dbs)\}), (Math,\{(201, Calculus)\}), (Philosophy,\{(302, Logic)\})\}$
102	Chris	$\{(CS,\{(200, PL), (202, Dbs), (301, AI)\}), (Philosophy,\{(302, Logic)\})\}$
103	Dinska	$\{(Math,\{(201, Calculus)\})\}$
105	Vince	$\{\}$

JSON (JavaScript Object Notation) objects

A **JSON** object (self-) describing a person:

```
{ "firstName": "John",
  "lastName": "Smith",
  "age": 25,
  "isAlive": true,
  "address": { "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100" },
  "phoneNumbers": [
    { "type": "home",
      "number": "212 555-1234" },
    { "type": "mobile",
      "number": "123 456-7890" } ],
  "children": [],
  "spouse": null
}
```

%

JSON **basic**, array, and null values

JSON objects are inductively defined using atomic, array, **null**, and object values:

- **Atomic values:**
 - **Number**: a signed decimal number (double-precision floating-point format)
 - **Strings**: a sequence of zero or more characters. Strings are delimited with double-quotation marks
 - **Boolean**: either of the values **true** or **false**
- **Array values:**
 - An **ordered list** of zero or more values, each of which may be of **any type**
 - Arrays use square bracket notation and array elements are comma-separated
- **null value**: An empty value, using the word **null**

JSON **object** values

- **Object value:**
 - An **unordered collection** of **key-value** pairs where the keys are strings
 - Objects are delimited with curly brackets and use commas to separate each pair, while within each pair the colon ‘:’ character separates the key from its value
 - Since objects are intended to represent associative arrays (maps), typically each key is unique within an object

XML documents

- Just like a JSON value, an **XML document** is a **self-describing** data object
- In contrast with the predefined type-structured nature of relations and nested relations, JSON values and XML documents are examples of **semi-structured** data
 - In JSON, the structure is revealed in the **keys** of the objects
 - In XML, the structure is revealed in the labeled **tags** of the document

Example: XML document

- The labeled tags (in red) provide the structure of the document
- The values in the document appear in green

```
<person>
  <firstName> John </firstName>
  <lastName> Smith </lastName>
  <age> 25 </age>
  <address>
    <streetAddress> 21 2nd Street </streetAddress>
    <city> New York </city>
    <state> NY </state>
    <postalCode> 10021 </postalCode>
  </address>
  <phoneNumber>
    <type> home </type>
    <number> 212 555-1234</number>
  </phoneNumber>
  <phoneNumber>
    <type> fax </type>
    <number> 646 555-4567 </number>
  </phoneNumber>
  <gender>
    <type> male</type>
  </gender>
</person>
```

- In the remainder of this lecture, we will no longer focus on XML

Review of populating a nested relation with an INSERT INTO Query

- We assume that data has been inserted into the Enroll relation
- We can then populate the studentGrades relation with an INSERT INTO statement with a query that use repeated grouping and array aggregation:

```
INSERT INTO          studentGrades
WITH E AS           (SELECT sid, grade, array_agg(row(cno)::courseType) AS courses
                      FROM Enroll e
                      GROUP BY (sid, grade)),
F AS               (SELECT sid, array_agg(row(grade, courses)::gradeCoursesType) AS gradeInfo
                      FROM E
                      GROUP BY(sid))
SELECT sid, gradeInfo
FROM F;
```

Populating a JSON objects relation with an INSERT INTO Query

- We assume that data has been inserted into the Enroll relation
- We create the following table of type JSONB
`CREATE TABLE jStudentGrades (studentInfo JSONB)`
- We then restructure the Enroll data and insert it into this table

```
INSERT INTO      jstudentGrades
WITH E AS        (SELECT e.sid, e.grade,
                        array_to_json(array_agg(json_build_object('cno',e.cno))) as courses
                     FROM Enroll e
                     GROUP BY (e.sid, e.grade)),
F AS            (SELECT json_build_object('sid', sid, 'gradeInfo',
                        array_to_json(array_agg(json_build_object('grade', grade, 'courses', courses)))) as studentInfo
                     FROM E
                     GROUP BY(sid))
SELECT *          FROM F;
```

Populating a JSON objects relation with an INSERT INTO Query

We obtain the following table consisting of 5 JSON objects:

F
{ "sid" : 100, "gradeInfo" : [{ "grade" : "A", "courses" : [[{ "cno" : 200},{ "cno" : 202}]],{ "grade" : "B", "courses" : [[{ "cno" : 201}]]}]}]
{ "sid" : 101, "gradeInfo" : [{ "grade" : "B", "courses" : [[{ "cno" : 200}]]}, { "grade" : "A", "courses" : [[{ "cno" : 201},{ "cno" : 302},{ "cno" : 202}]],{ "grade" : "C", "courses" : [[{ "cno" : 301}]]}]}]
{ "sid" : 102, "gradeInfo" : [{ "grade" : "A", "courses" : [[{ "cno" : 202},{ "cno" : 302}]],{ "grade" : "B", "courses" : [[{ "cno" : 200},{ "cno" : 301}]]}]}]
{ "sid" : 103, "gradeInfo" : [{ "grade" : "A", "courses" : [[{ "cno" : 201}]]}]}]
{ "sid" : 104, "gradeInfo" : [{ "grade" : "D", "courses" : [[{ "cno" : 201}]]}]}]

JSON versus Nested Relation Construction Operators

```
INSERT INTO jstudentGrades
WITH E AS
    (SELECT sid, grade,
        array_to_json(array_agg(json_build_object('cno',cno))) as courses
     FROM Enroll
      GROUP BY (sid, grade)),
F AS
    SELECT json_build_object('sid', sid, 'gradeInfo',
        array_to_json(array_agg(json_build_object('grade', grade, 'courses', courses)))) as studentInfo
     FROM E
      GROUP BY (sid))
SELECT *
  FROM F;
```

Operator	JSON	Operator	Nested Relations
object construction	<code>json_build_object()</code>	row construction	<code>row()</code>
array construction	<code>array_to_json(array_agg())</code>	array construction	<code>array_agg()</code>

Example: Querying the JSON relation

- “Find the grade information of each student who received an ‘A’ in some course.”

```
SELECT      sg.studentInfo -> 'sid', sg.studentInfo -> 'gradeInfo'  
FROM        jstudentGrades sg,  
           jsonb_array_elements(sg.studentInfo -> 'gradeInfo') g  
WHERE       g -> 'grade' = 'A';
```

Operator extraction	JSON	Operator extraction	Nested Relations
	jsonb_array_elements()		UNNEST()

- Notice the similarity with the query expressed on the `studentGrades` nested relation

```
SELECT      sid, gradeInfo  
FROM        studentGrades sg, UNNEST(sg.gradeInfo) g  
WHERE       g.grade = 'A'
```

Example: Querying the JSON relation

- “Find the grade information of each student who received an ‘A’ in some course.”

```
SELECT      sg.studentInfo -> 'sid', sg.studentInfo -> 'gradeInfo'  
FROM        jstudentGrades sg,  
           jsonb_array_elements(sg.studentInfo -> 'gradeInfo') g  
WHERE       g -> 'grade' = ' "A" ';
```

- We obtain the following result:

```
{ "sid" : 100,  
  "gradeInfo" : [{ "grade" : "A", "courses" : [{ "cno" : 200},{ "cno" : 202}],{ "grade" : "B", "courses" : [{ "cno" : 201}]}]}  
  
{ "sid" : 101,  
  "gradeInfo" : [{ "grade" : "B", "courses" : [{ "cno" : 200}]),{ "grade" : "A",  
  "courses" : [{ "cno" : 201},{ "cno" : 302},{ "cno" : 202}],{ "grade" : "C", "courses" : [{ "cno" : 301}]}]}  
  
{ "sid" : 102,  
  "gradeInfo" : [{ "grade" : "A", "courses" : [{ "cno" : 202},{ "cno" : 302}],{ "grade" : "B",  
  "courses" : [{ "cno" : 200},{ "cno" : 301}]}]}  
  
{ "sid" : 103, "gradeInfo" : [{ "grade" : "A", "courses" : [{ "cno" : 201}]}]}
```

Example: Querying the JSON relation

- “Find the grade information of each student who enrolled in course 301.”

```
SELECT    sg.studentInfo -> 'sid', sg.studentInfo -> 'gradeInfo'  
FROM      jstudentGrades sg,  
          jsonb_array_elements(sg.studentInfo -> 'gradeInfo') g,  
          jsonb_array_elements(g -> 'courses') c  
WHERE     (c -> 'cno')::jsonb = '301';
```

- Notice the similarity with the query expressed for the `studentGrades` nested relation

```
SELECT    sid, gradeInfo  
FROM      studentGrades sg,  
          UNNEST(sg.gradeInfo) g,  
          UNNEST(g.courses) c  
WHERE     c.cno = 301;
```

Example: Querying the JSON relation

- “For each student, find the set of courses in which he or she is enrolled.”

```
SELECT      sg.studentInfo -> 'sid' as sid , array_to_json(array_agg(c -> 'cno')) as courses
FROM        jstudentGrades sg,
            jsonb_array_elements(sg.studentInfo -> 'gradeInfo') g,
            jsonb_array_elements(g -> 'courses') c
GROUP BY    (sg.studentInfo -> 'sid')
```

- The result is as follows:

sid	courses
100	[200,201,202]
101	[200,201,202,301,302]
102	[200,202,301,302]
103	[201]
104	[201]

Example: Querying the JSON relation

- “For each student, find the set of courses in which he or she is enrolled.”

```
SELECT      sg.studentInfo -> 'sid' as sid , array_to_json(array_agg(c -> 'cno')) as courses
FROM        jstudentGrades sg,
            jsonb_array_elements(sg.studentInfo -> 'gradeInfo') g,
            jsonb_array_elements(g -> 'courses') c
GROUP BY    (sg.studentInfo -> 'sid')
```

- Notice the similarity with the query expressed for the `studentGrades` nested relation

```
SELECT      sid, array_agg(c.cno) as courses
FROM        studentGrades sg,
            UNNEST(sg.gradeInfo) g,
            UNNEST(g.courses) c
GROUP BY    (sg.sid)
```

MongoDb

- MongoDb is a semi-structured data and programming model to store, search, and manipulate collections of documents
- A collection is an array of documents represented as JSON objects:

```
db.jstudentGrades
[  
  { "sid" : 100,  
    "gradeInfo" : [{ "grade" : "A", "courses" : [{ "cno" : 200},{ "cno" : 202}]],{ "grade" : "B", "courses" : [{ "cno" : 201}],  
  { "sid" : 101,  
    "gradeInfo" : [{ "grade" : "B", "courses" : [{ "cno" : 200}]],{ "grade" : "A",  
      "courses" : [{ "cno" : 201},{ "cno" : 302},{ "cno" : 202}]],{ "grade" : "C", "courses" : [{ "cno" : 301}]}}},  
  { "sid" : 102,  
    "gradeInfo" : [{ "grade" : "A", "courses" : [{ "cno" : 202},{ "cno" : 302}]],{ "grade" : "B",  
      "courses" : [{ "cno" : 200},{ "cno" : 301}]}}},  
  { "sid" : 103, "gradeInfo" : [{ "grade" : "A", "courses" : [{ "cno" : 201}]}}},  
  { "sid" : 104, "gradeInfo" : [{ "grade" : "D", "courses" : [{ "cno" : 201}]}}}  
]
```

MongoDb (Queries)

- Queries and updates typically take the following form

```
db.<collection>.<method>(<filter>, <options>)
```

db	name of the database
collection	name of a collection in db
method	operation on (objects in) collection
filter	conditions that selects objects in collection
options	each method has certain options for what is will do with documents that match the filter condition

MongoDb (Path queries)

- Find the student grade info for student with sid = '102':

```
db.<jstudentGrades>.find({'sid': 102})
```

- Find the student grade info for students who received an 'A' in some course:

```
db.<jstudentGrades>.find({'gradeInfo.grade': 'A'})
```

Here '**gradeInfo.grade**' is a path.

- Find the student grade info for students who took course 301:

```
db.<jstudentGrades>.find({'gradeInfo.courses.cno': 301})
```

Here '**gradeInfo.courses.cno**' is a path.

NoSQL Databases and Big Data Storage Systems

Introduction

- NoSQL
 - Not only SQL
- Most NoSQL systems are distributed databases or distributed storage systems
 - Focus on semi-structured data storage, high performance, availability, data replication, and scalability

Introduction (cont'd.)

- NoSQL systems focus on storage of “big data”
- Typical applications that use NoSQL
 - Social media
 - Web links
 - User profiles
 - Marketing and sales
 - Posts and tweets
 - Road maps and spatial data
 - Email

24.1 Introduction to NoSQL Systems

- BigTable
 - Google's proprietary NoSQL system
 - Column-based or wide column store
- DynamoDB (Amazon)
 - Key-value data store
- Cassandra (Facebook)
 - Uses concepts from both key-value store and column-based systems

Introduction to NoSQL Systems (cont'd.)

- MongoDB and CouchDB
 - Document stores
- Neo4J and GraphBase
 - Graph-based NoSQL systems
- OrientDB
 - Combines several concepts
- Database systems classified on the object model
 - Or native XML model

Introduction to NoSQL Systems (cont'd.)

- NoSQL characteristics related to distributed databases and distributed systems
 - Scalability
 - Availability, replication, and eventual consistency
 - Sharding of files
 - High performance data access

Introduction to NoSQL Systems (cont'd.)

- NoSQL characteristics related to data models and query languages
 - Schema not required
 - Less powerful query languages
 - Versioning

Introduction to NoSQL Systems (cont'd.)

- Categories of NoSQL systems
 - Document-based NoSQL systems
 - NoSQL key-value stores
 - Column-based or wide column NoSQL systems
 - Graph-based NoSQL systems
 - Object/object relational databases
 - XML databases

Basic Concepts: Sharding

- Sharding is the process of horizontally partitioning a large dataset into a collection of smaller, more manageable datasets called shards.
- The shards are distributed across multiple nodes, where a node is a server or a machine.
- Each shard is stored on a separate node and each node is responsible for only the data stored on it.
- Each shard shares the same schema, and all shards collectively represent the complete dataset.
- Since each node is responsible for only a part of the whole dataset, read/write times are greatly improved.

Basic Concepts: Replication

- Replication stores multiple copies of a dataset, known as *replicas*, on multiple nodes.
- Replication provides scalability and availability due to the fact that the same data is replicated on various nodes.
- Fault tolerance is also achieved since data redundancy ensures that data is not lost when an individual node fails.
- There are two different methods that are used to implement replication:
 - master-slave
 - peer-to-peer

Basic Concepts: Master Slave Replication

- During master-slave replication, nodes are arranged in a master-slave configuration, and all data is written to a master node.
- Once saved, the data is replicated over to multiple slave nodes.
- All external write requests, including insert, update and delete, occur on the master node, whereas read requests can be fulfilled by any slave node.
- One concern with master-slave replication is read inconsistency, which can be an issue if a slave node is read prior to an update to the master being copied to it.
- To ensure read consistency, a voting system can be implemented where a read is declared consistent if the majority of the slaves contain the same version of the record.

Basic Concepts: Peer to Peer Replication

- With peer-to-peer replication, all nodes operate at the same level.
- Each node, known as a peer, is equally capable of handling reads and writes.
- Each write is copied to all peers.
- Peer-to-peer replication is prone to write inconsistencies that occur as a result of a simultaneous update of the same data across multiple peers.
 - Concurrency control can lock the record being updated. While this ensures consistency, availability is affected
- Like master-slave replication, reads can be inconsistent during the time period when some of the peers have completed their updates while others perform their updates.
- Reads eventually become consistent when the updates have been executed on all peers.

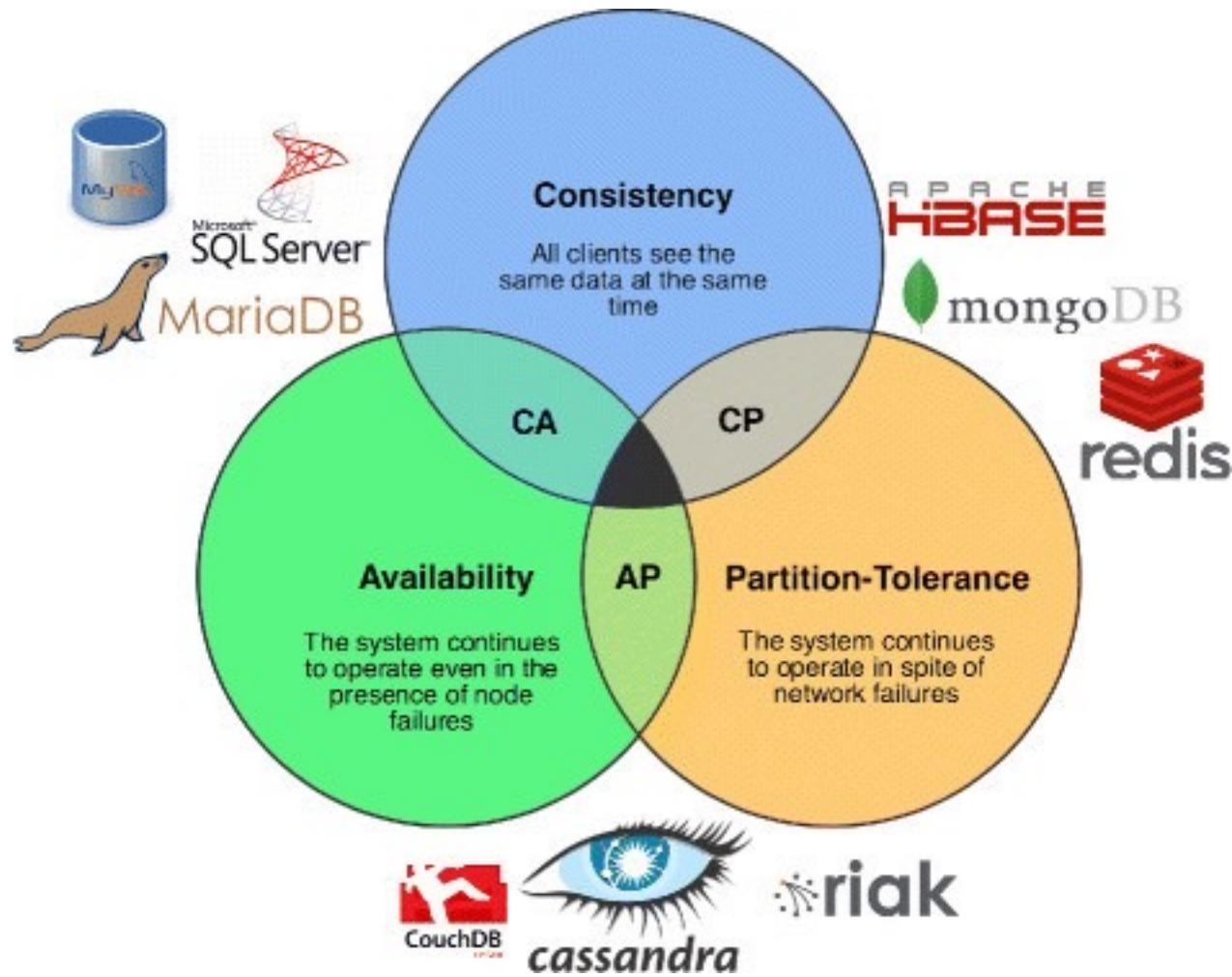
24.2 The CAP Theorem

- Consistency, availability, and partition tolerance
 - Consistency – A read from any node results in the same data across multiple nodes.
 - *Availability* – A read/write request will always be acknowledged in the form of a success or a failure
 - *Partition tolerance* – The database system can tolerate communication outages that split the cluster into multiple silos and can still service read/write requests
- Not possible to guarantee all three simultaneously
 - In distributed system with data replication

The CAP Theorem (cont'd.)

- CAP theorem demonstrates why only two of the three properties of the CAP theorem are simultaneously supportable.
 - If consistency (C) and availability (A) are required, available nodes need to communicate to ensure consistency (C). So, partition tolerance (P) is not possible.
 - If consistency (C) and partition tolerance (P) are required, nodes cannot remain available
 - If availability (A) and partition tolerance (P) are required, then consistency (C) is not possible because of the data communication requirement between the nodes.

The CAP Theorem (cont'd.)



24.3 Document-Based NoSQL Systems and MongoDB

- Document stores
 - Collections of similar documents
- Individual documents resemble complex objects or XML documents
 - Documents are self-describing
 - Can have different data elements
- Documents can be specified in various formats
 - XML
 - JSON

Document-Based NoSQL Systems

Key	Document
1001	{ "CustomerID": 99, "OrderItems": [{ "ProductID": 2010, "Quantity": 2, "Cost": 520 }, { "ProductID": 4365, "Quantity": 1, "Cost": 18 }], "OrderDate": "04/01/2017" }
1002	{ "CustomerID": 220, "OrderItems": [{ "ProductID": 1285, "Quantity": 1, "Cost": 120 }], "OrderDate": "05/08/2017" }

<https://docs.microsoft.com/en-us/azure/architecture/data-guide/big-data/non-relational-data>

MongoDB Data Model

- Documents stored in binary JSON (BSON) format
- Individual documents stored in a collection
- Example command
 - First parameter specifies name of the collection
 - Collection options include limits on size and number of documents
- db.createCollection("project", { capped : true, size : 1310720, max : 500 })
- Each document in collection has unique ObjectId field called _id

MongoDB Data Model (cont'd.)

- A collection does not have a schema
 - Structure of the data fields in documents chosen based on how documents will be accessed
- Document creation using insert operation

```
db.<collection_name>.insert(<document(s)>)
```

- Document deletion using remove operation

```
db.<collection_name>.remove(<condition>)
```

(a) project document with an array of embedded workers:

```
{  
    _id: "P1",  
    Pname: "ProductX",  
    Plocation: "Bellaire",  
    Workers: [  
        { Ename: "John Smith",  
         Hours: 32.5  
        },  
        { Ename: "Joyce English",  
         Hours: 20.0  
        }  
    ]  
};
```

MongoDB Distributed Systems

Characteristics

- Replication in MongoDB
 - Concept of replica set to create multiple copies on different nodes
 - Variation of master-slave approach
 - Primary copy, secondary copy, and arbiter
 - Arbiter participates in elections to select new primary if needed

MongoDB Distributed Systems

Characteristics (cont'd.)

- Replication in MongoDB (cont'd.)
 - All write operations applied to the primary copy and propagated to the secondaries
 - User can choose read preference
 - Read requests can be processed at any replica
- Sharding in MongoDB
 - Horizontal partitioning divides the documents into disjoint partitions (shards)
 - Allows adding more nodes as needed
 - Shards stored on different nodes to achieve load balancing

MongoDB Distributed Systems

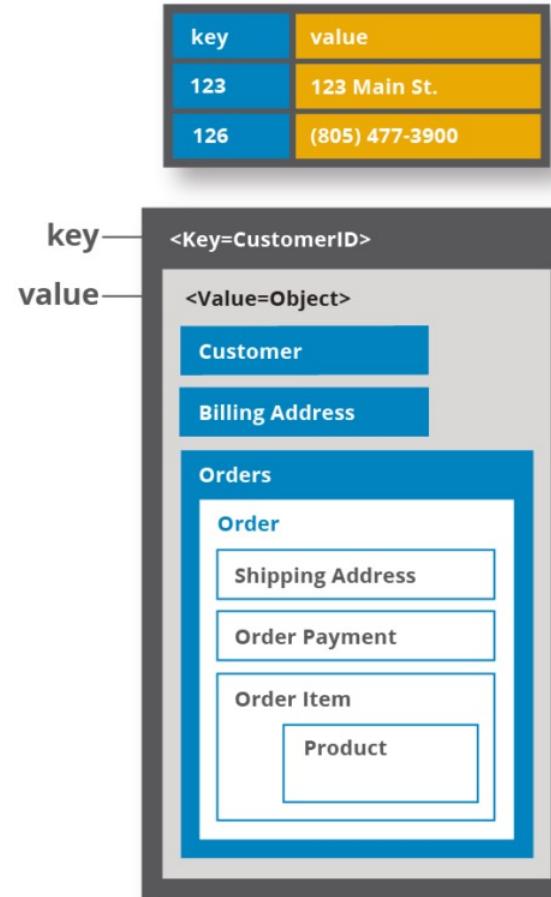
Characteristics (cont'd.)

- Sharding in MongoDB (cont'd.)
 - Partitioning field (shard key) must exist in every document in the collection
 - Must have an index
 - Range partitioning
 - Creates chunks by specifying a range of key values
 - Hash partitioning
 - Partitioning based on the hash values of each shard key

24.4 NoSQL Key-Value Stores

- Key-value stores focus on high performance, availability, and scalability
 - Can store structured, unstructured, or semi-structured data
- Key: unique identifier associated with a data item
 - Used for fast retrieval
- Value: the data item itself
 - Can be string or array of bytes
 - Application interprets the structure
- No query language

NoSQL Key-Value Stores



<https://hazelcast.com/glossary/key-value-store/>

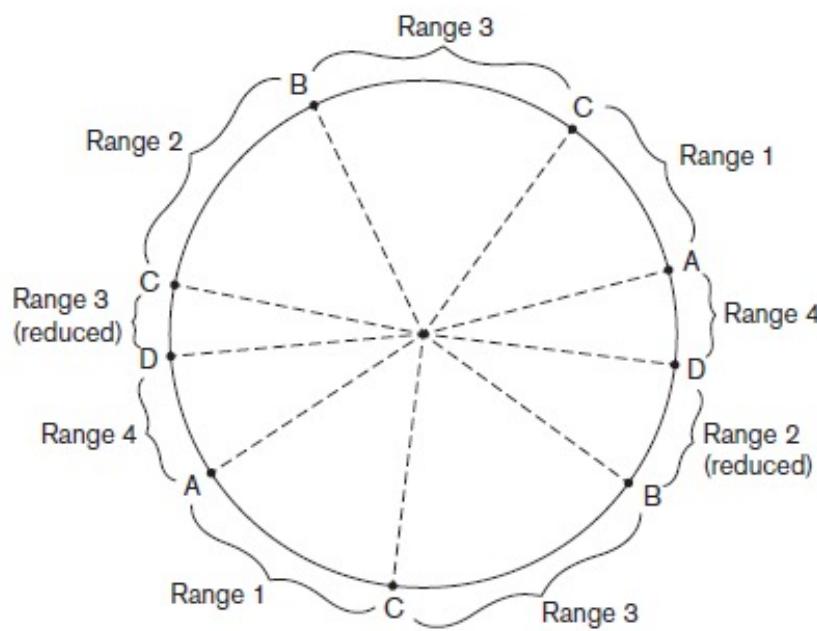
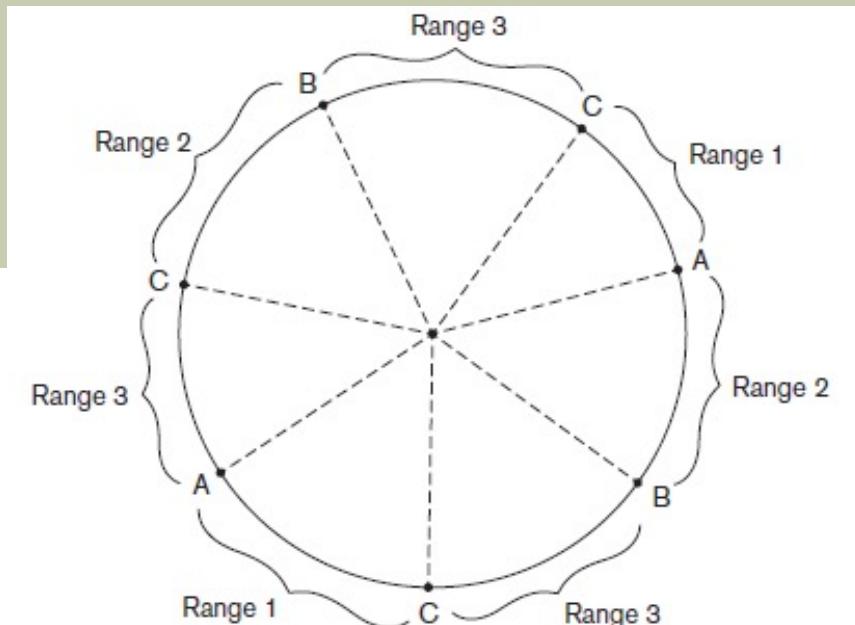
DynamoDB Overview

- DynamoDB part of Amazon's Web Services/SDK platforms
 - Proprietary
- Table holds a collection of self-describing items
- Item consists of attribute-value pairs
 - Attribute values can be single or multi-valued
- Primary key used to locate items within a table
 - Can be single attribute or pair of attributes

Voldemort Key-Value Distributed Data Store

- Voldemort: open-source key-value system similar to DynamoDB
- Voldemort features
 - Simple basic operations (get, put, and delete)
 - High-level formatted data values
 - Consistent hashing for distributing (key, value) pairs

Figure 24.2 Example of consistent hashing (a) Ring having three nodes A, B, and C, with C having greater capacity. The $h(K)$ values that map to the circle points in range 1 have their (k, v) items stored in node A, range 2 in node B, range 3 in node C (b) Adding a node D to the ring. Items in range 4 are moved to the node D from node B (range 2 is reduced) and node C (range 3 is reduced)



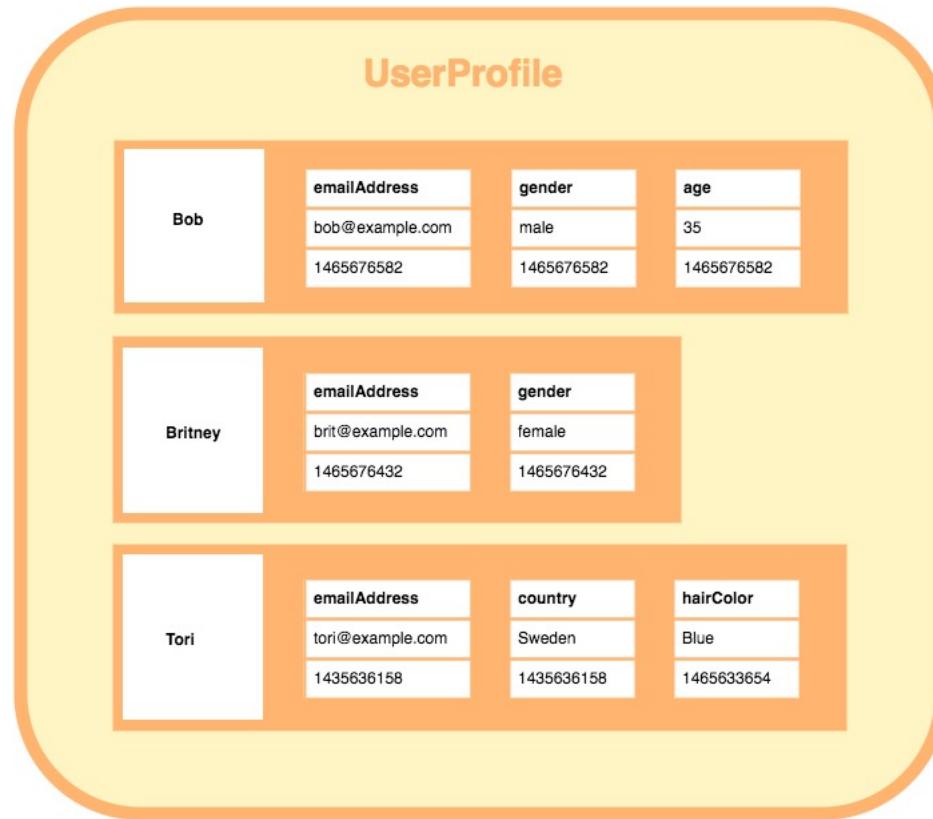
Examples of Other Key-Value Stores

- Oracle key-value store
 - Oracle NoSQL Database
- Redis key-value cache and store
 - Caches data in main memory to improve performance
 - Offers master-slave replication and high availability
 - Offers persistence by backing up cache to disk
- Apache Cassandra
 - Offers features from several NoSQL categories
 - Used by Facebook and others

24.5 Column-Based or Wide Column NoSQL Systems

- BigTable: Google's distributed storage system for big data
 - Used in Gmail
 - Uses Google File System for data storage and distribution
- Apache Hbase a similar, open-source system
 - Uses Hadoop Distributed File System (HDFS) for data storage
 - Can also use Amazon's Simple Storage System (S3)

Column-Based NoSQL Systems



<https://database.guide/what-is-a-column-store-database/>

Hbase Data Model and Versioning

- Data organization concepts
 - Namespaces
 - Tables
 - Column families
 - Column qualifiers
 - Columns
 - Rows
 - Data cells
- Data is self-describing

Hbase Data Model and Versioning (cont'd.)

- Namespace is collection of tables
- Table associated with one or more column families
- Column qualifiers can be dynamically specified as new table rows are created and inserted
- Each row in a table has a unique row key
- Cell holds a basic data item
- HBase stores multiple versions of data items
 - Timestamp associated with each version

(a) creating a table:

```
create 'EMPLOYEE', 'Name', 'Address', 'Details'
```

(b) Inserting some row data In the EMPLOYEE table:

```
put 'EMPLOYEE', 'row1', 'Name:Fname', 'John'  
put 'EMPLOYEE', 'row1', 'Name:Lname', 'Smith'  
put 'EMPLOYEE', 'row1', 'Name:Nickname', 'Johnny'  
put 'EMPLOYEE', 'row1', 'Details:Job', 'Engineer'  
put 'EMPLOYEE', 'row1', 'Details:Review', 'Good'  
put 'EMPLOYEE', 'row2', 'Name:Fname', 'Alicia'  
put 'EMPLOYEE', 'row2', 'Name:Lname', 'Zelaya'  
put 'EMPLOYEE', 'row2', 'Name:MName', 'Jennifer'  
put 'EMPLOYEE', 'row2', 'Details:Job', 'DBA'  
put 'EMPLOYEE', 'row2', 'Details:Supervisor', 'James Borg'  
put 'EMPLOYEE', 'row3', 'Name:Fname', 'James'  
put 'EMPLOYEE', 'row3', 'Name:Minit', 'E'  
put 'EMPLOYEE', 'row3', 'Name:Lname', 'Borg'  
put 'EMPLOYEE', 'row3', 'Name:Suffix', 'Jr.'  
put 'EMPLOYEE', 'row3', 'Details:Job', 'CEO'  
put 'EMPLOYEE', 'row3', 'Details:Salary', '1,000,000'
```

(c) Some Hbase basic CRUD operations:

Creating a table: create <tablename>, <column family>, <column family>, ...

Inserting Data: put <tablename>, <rowid>, <column family>:<column qualifier>, <value>

Reading Data (all data in a table): scan <tablename>

Retrieve Data (one item): get <tablename>,<rowid>

Figure 24.3 Examples in Hbase (a) Creating a table called EMPLOYEE with three column families: Name, Address, and Details (b) Inserting some in the EMPLOYEE table; different rows can have different self-describing column qualifiers (Fname, Lname, Nickname, Mname, Minit, Suffix, ... for column family Name; Job, Review, Supervisor, Salary for column family Details). (c) Some CRUD operations of Hbase

Hbase Crud Operations

- Provides only low-level CRUD (create, read, update, delete) operations
- Application programs implement more complex operations
- Create
 - Creates a new table and specifies one or more column families associated with the table
- Put
 - Inserts new data or new versions of existing data items

Hbase Crud Operations (cont'd.)

- Get
 - Retrieves data associated with a single row
- Scan
 - Retrieves all the rows

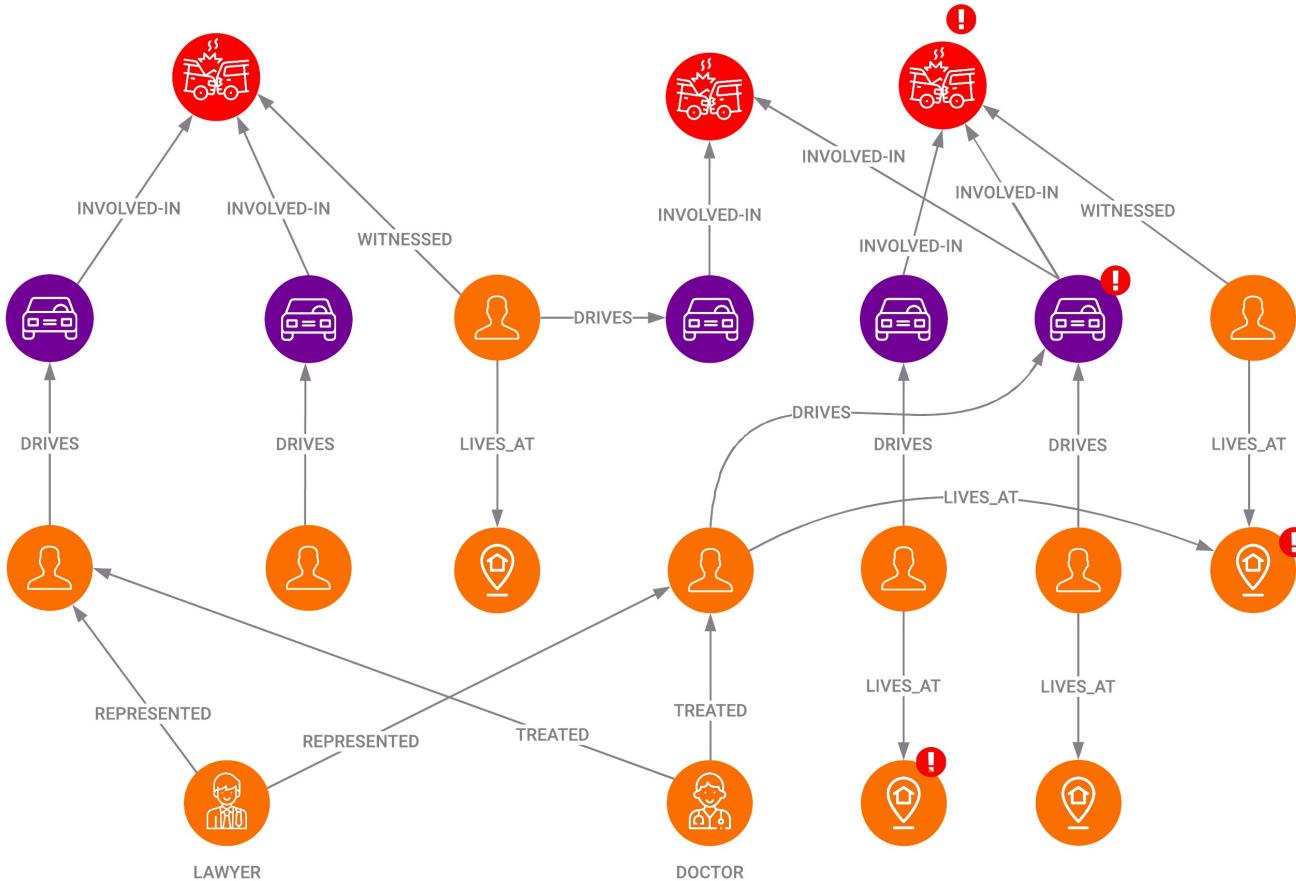
Hbase Storage and Distributed System Concepts

- Each Hbase table divided into several regions
 - Each region holds a range of the row keys in the table
 - Row keys must be lexicographically ordered
 - Each region has several stores
 - Column families are assigned to stores
- Regions assigned to region servers for storage
 - Master server responsible for monitoring the region servers
- Hbase uses Apache Zookeeper and HDFS

24.6 NoSQL Graph Databases and Neo4j

- Graph databases
 - Data represented as a graph
 - Collection of vertices (nodes) and edges
 - Possible to store data associated with both individual nodes and individual edges
- Neo4j
 - Open source system
 - Uses concepts of nodes and relationships

NoSQL Graph Databases



<https://memgraph.com/blog/why-your-business-should-use-a-graph-database>

Neo4j (cont'd.)

- Nodes can have labels
 - Zero, one, or several
- Both nodes and relationships can have properties
- Each relationship has a start node, end node, and a relationship type
- Properties specified using a map pattern
- Somewhat similar to ER/EER concepts

Neo4j (cont'd.)

- Creating nodes in Neo4j
 - CREATE command
 - Part of high-level declarative query language Cypher
 - Node label can be specified when node is created
 - Properties are enclosed in curly brackets

Neo4j (cont'd.)

- (a) creating some nodes for the COMPANY data (from Figure 5.6):

```
CREATE (e1: EMPLOYEE, {Empid: '1', Lname: 'Smith', Fname: 'John', Minit: 'B'})  
CREATE (e2: EMPLOYEE, {Empid: '2', Lname: 'Wong', Fname: 'Franklin'})  
CREATE (e3: EMPLOYEE, {Empid: '3', Lname: 'Zelaya', Fname: 'Alicia'})  
CREATE (e4: EMPLOYEE, {Empid: '4', Lname: 'Wallace', Fname: 'Jennifer', Minit: 'S'})  
...  
CREATE (d1: DEPARTMENT, {Dno: '5', Dname: 'Research'})  
CREATE (d2: DEPARTMENT, {Dno: '4', Dname: 'Administration'})  
...  
CREATE (p1: PROJECT, {Pno: '1', Pname: 'ProductX'})  
CREATE (p2: PROJECT, {Pno: '2', Pname: 'ProductY'})  
CREATE (p3: PROJECT, {Pno: '10', Pname: 'Computerization'})  
CREATE (p4: PROJECT, {Pno: '20', Pname: 'Reorganization'})  
...  
CREATE (loc1: LOCATION, {Lname: 'Houston'})  
CREATE (loc2: LOCATION, {Lname: 'Stafford'})  
CREATE (loc3: LOCATION, {Lname: 'Bellaire'})  
CREATE (loc4: LOCATION, {Lname: 'Sugarland'})  
...
```

Figure 24.4 Examples in Neo4j using the Cypher language (a) Creating some nodes

Neo4j (cont'd.)

- (b) creating some relationships for the COMPANY data (from Figure 5.6):

```
CREATE (e1) - [ : WorksFor ] -> (d1)
```

```
CREATE (e3) - [ : WorksFor ] -> (d2)
```

...

```
CREATE (d1) - [ : Manager ] -> (e2)
```

```
CREATE (d2) - [ : Manager ] -> (e4)
```

...

```
CREATE (d1) - [ : LocatedIn ] -> (loc1)
```

```
CREATE (d1) - [ : LocatedIn ] -> (loc3)
```

```
CREATE (d1) - [ : LocatedIn ] -> (loc4)
```

```
CREATE (d2) - [ : LocatedIn ] -> (loc2)
```

...

```
CREATE (e1) - [ : WorksOn, {Hours: '32.5'} ] -> (p1)
```

```
CREATE (e1) - [ : WorksOn, {Hours: '7.5'} ] -> (p2)
```

```
CREATE (e2) - [ : WorksOn, {Hours: '10.0'} ] -> (p1)
```

```
CREATE (e2) - [ : WorksOn, {Hours: 10.0} ] -> (p2)
```

```
CREATE (e2) - [ : WorksOn, {Hours: '10.0'} ] -> (p3)
```

```
CREATE (e2) - [ : WorksOn, {Hours: 10.0} ] -> (p4)
```

...

Figure 24.4 (cont'd.) Examples in Neo4j using the Cypher language
(b) Creating some relationships

Neo4j (cont'd.)

- Path
 - Traversal of part of the graph
 - Typically used as part of a query to specify a pattern
- Schema optional in Neo4j
- Indexing and node identifiers
 - Users can create for the collection of nodes that have a particular label
 - One or more properties can be indexed

The Cypher Query Language of Neo4j

- Cypher query made up of clauses
- Result from one clause can be the input to the next clause in the query

The Cypher Query Language of Neo4j (cont'd.)

(c) Basic simplified syntax of some common Cypher clauses:

Finding nodes and relationships that match a pattern: MATCH <pattern>

Specifying aggregates and other query variables: WITH <specifications>

Specifying conditions on the data to be retrieved: WHERE <condition>

Specifying the data to be returned: RETURN <data>

Ordering the data to be returned: ORDER BY <data>

Limiting the number of returned data items: LIMIT <max number>

Creating nodes: CREATE <node, optional labels and properties>

Creating relationships: CREATE <relationship, relationship type and optional properties>

Deletion: DELETE <nodes or relationships>

Specifying property values and labels: SET <property values and labels>

Removing property values and labels: REMOVE <property values and labels>

Figure 24.4 (cont'd.) Examples in Neo4j using the Cypher language

(c) Basic syntax of Cypher queries

The Cypher Query Language of Neo4j (cont'd.)

(d) Examples of simple Cypher queries:

1. MATCH (d : DEPARTMENT {Dno: '5'}) – [: LocatedIn] → (loc)
RETURN d.Dname , loc.Lname
2. MATCH (e: EMPLOYEE {Empid: '2'}) – [w: WorksOn] → (p)
RETURN e.Ename , w.Hours, p.Pname
3. MATCH (e) – [w: WorksOn] → (p: PROJECT {Pno: 2})
RETURN p.Pname, e.Ename , w.Hours
4. MATCH (e) – [w: WorksOn] → (p)
RETURN e.Ename , w.Hours, p.Pname
ORDER BY e.Ename
5. MATCH (e) – [w: WorksOn] → (p)
RETURN e.Ename , w.Hours, p.Pname
ORDER BY e.Ename
LIMIT 10
6. MATCH (e) – [w: WorksOn] → (p)
WITH e, COUNT(p) AS numOfprojs
WHERE numOfprojs > 2
RETURN e.Ename , numOfprojs
ORDER BY numOfprojs
7. MATCH (e) – [w: WorksOn] → (p)
RETURN e , w, p
ORDER BY e.Ename
LIMIT 10
8. MATCH (e: EMPLOYEE {Empid: '2'})
SET e.Job = 'Engineer'

Figure 24.4 (cont'd.) Examples in Neo4j using the Cypher language
(d) Examples of Cypher queries

Neo4j Interfaces and Distributed System Characteristics

- Enterprise edition versus community edition
 - Enterprise edition supports caching, clustering of data, and locking
- Graph visualization interface
 - Subset of nodes and edges in a database graph can be displayed as a graph
 - Used to visualize query results
- Master-slave replication
- Caching
- Logical logs

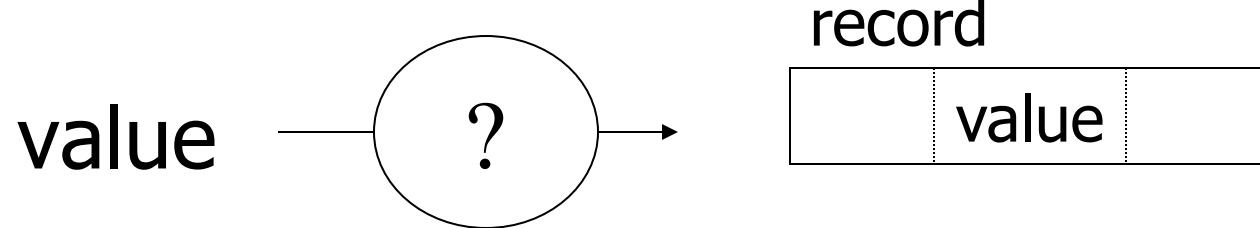
24.7 Summary

- NoSQL systems focus on storage of “big data”
- General categories
 - Document-based
 - Key-value stores
 - Column-based
 - Graph-based
 - Some systems use techniques spanning two or more categories
- Consistency paradigms
- CAP theorem

Notes: Indexing General concepts

Based on lectures by Hector
Garcia-Molina

Indexing & Hashing



Topics

- Conventional indexes
- B-trees
- Hashing schemes

Sequential File

10	
20	

30	
40	

50	
60	

70	
80	

90	
100	

Given a key, a sequential file store records in contiguous block sorted on key values

Dense Index

10	
20	
30	
40	

50	
60	
70	
80	

90	
100	
110	
120	

Sequential File

10	
20	

30	
40	

50	
60	

70	
80	

90	
100	

Notice that the dense index is much smaller than the sequential file which hold the data
Each arrow is a block address

Sparse Index

10	
30	
50	
70	

90	
110	
130	
150	

170	
190	
210	
230	

Sequential File

10	
20	

30	
40	

50	
60	

70	
80	

90	
100	

In a sparse index, an index pointer points to the block in the sequential file that hold all the records whose key values are in the range of 2 consecutive index (key) values in the sparse index

Sparse 2nd level

10	-
90	
170	
250	

330	-
410	
490	
570	

Sparse 1st level

10	-
30	
50	
70	

90	-
110	
130	
150	

170	-
190	
210	
230	

Sequential File

10	-
20	

30	-
40	

50	-
60	

70	-
80	

90	-
100	

Notice that the sparse index on the 2nd level is much smaller than the sparse index on the 1st level

- Comment:
The blocks in the sequential data file or in at a level of the index may be in contiguous memory or may be in blocks that are in a (chained) list

Inserts in a list is much easier (more efficient) in a list.

Inserts in contiguous memory may require a shift that is linear in the size of the data file or an index level

Question:

- Can we build a dense, 2nd level index for a dense index?

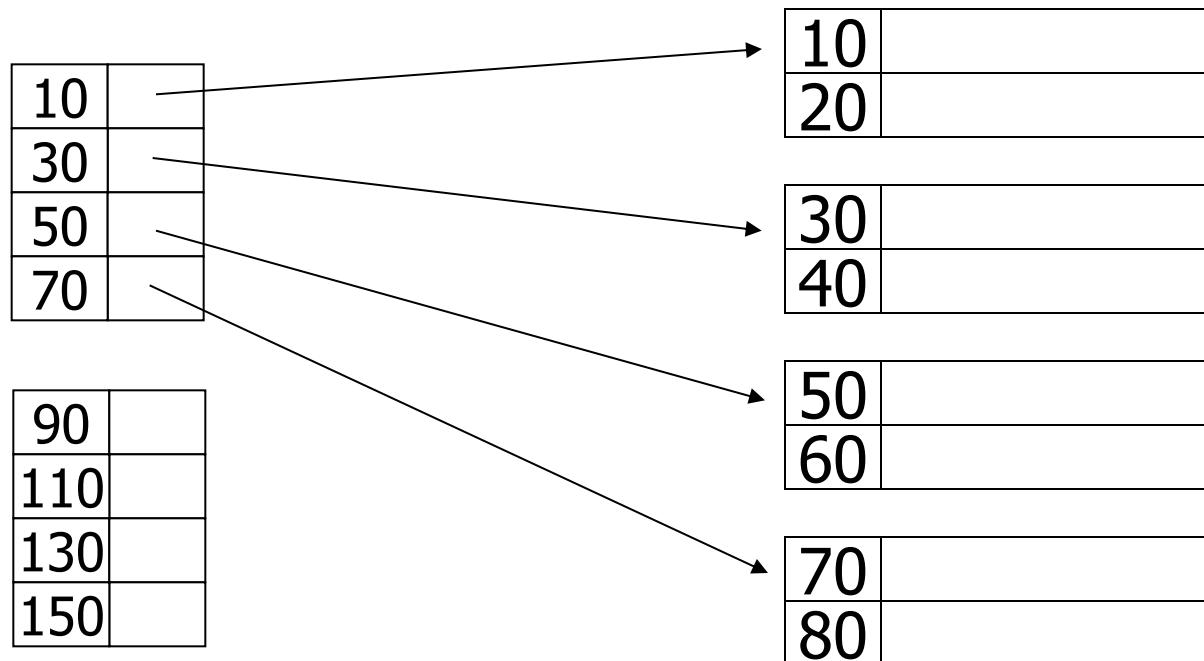
Sparse vs. Dense Tradeoff

- Sparse: Less index space per record
can keep more of index in memory
- Dense: Can tell if any record exists
without accessing file

Recap of concepts and terms related to indexing

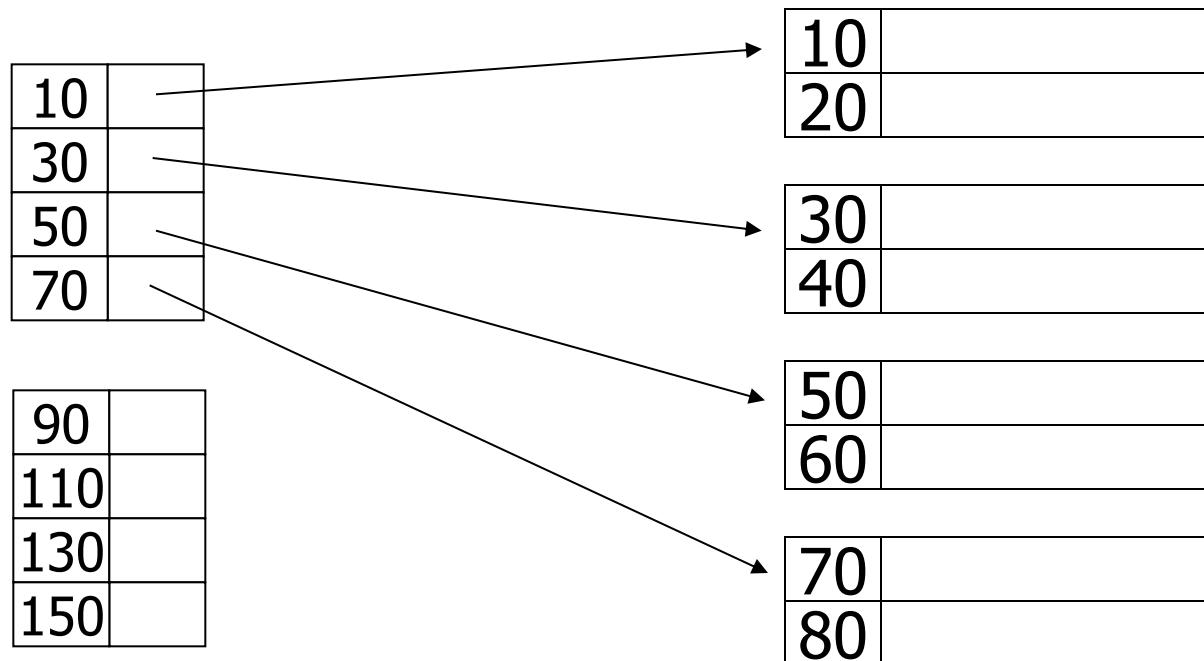
- Sequential file
- Search key (\neq primary key)
- Primary index
- Secondary index
- Dense index
- Sparse index
- Multi-level index

Deletion from sparse index



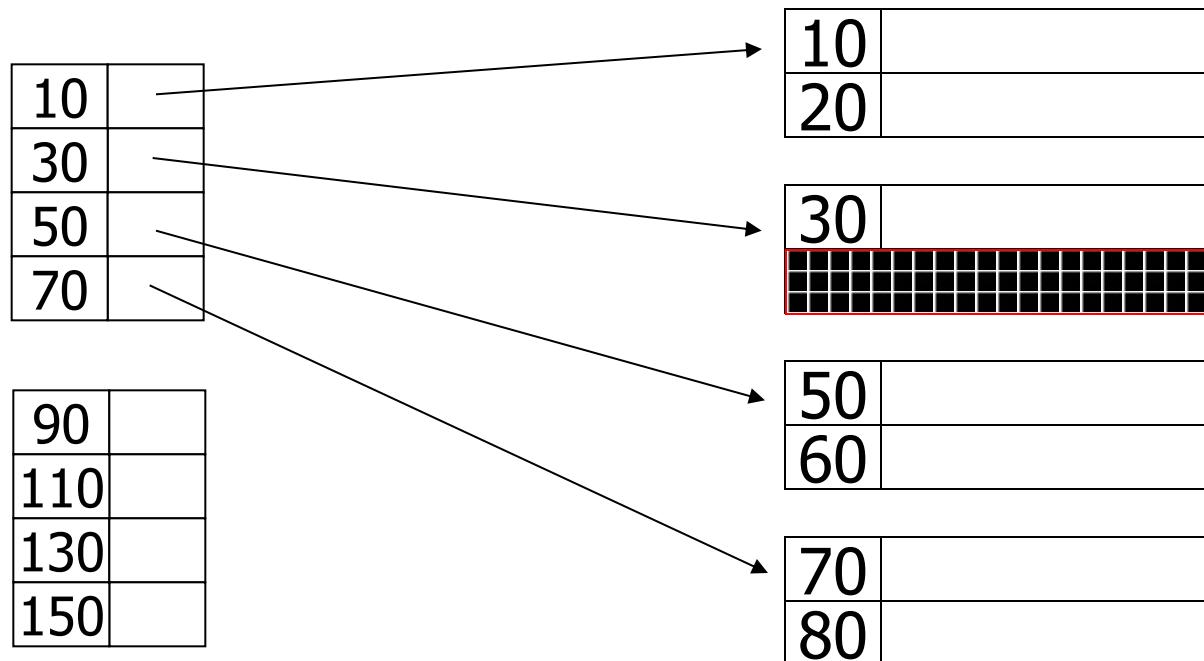
Deletion from sparse index

– delete record 40



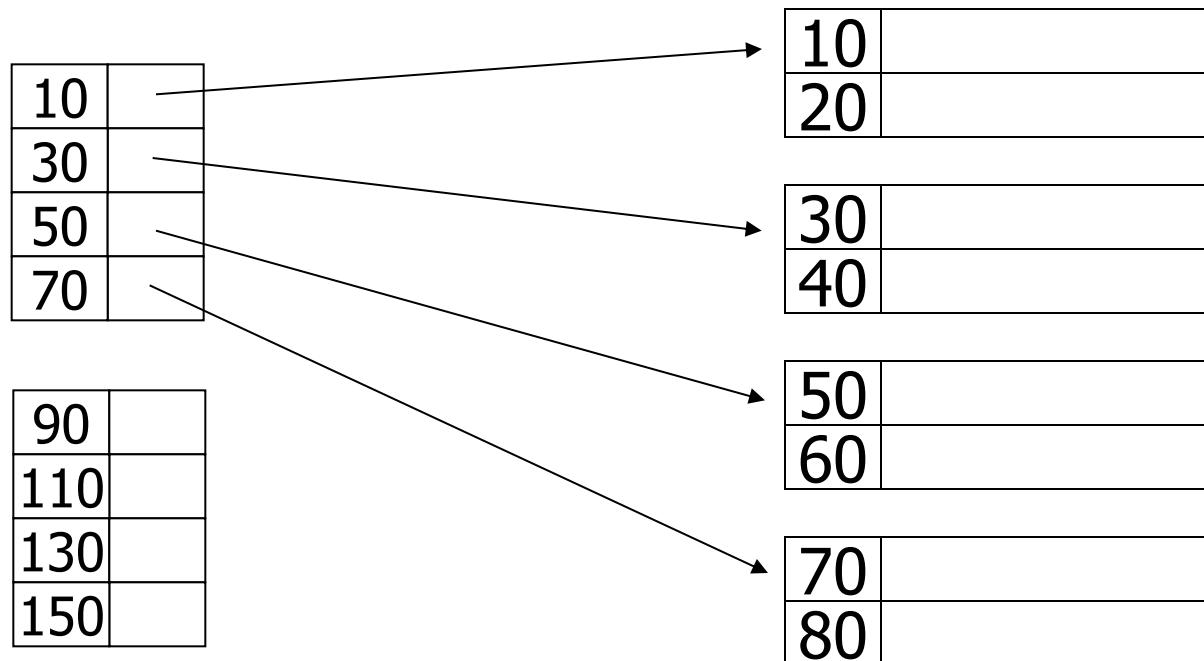
Deletion from sparse index

– delete record 40



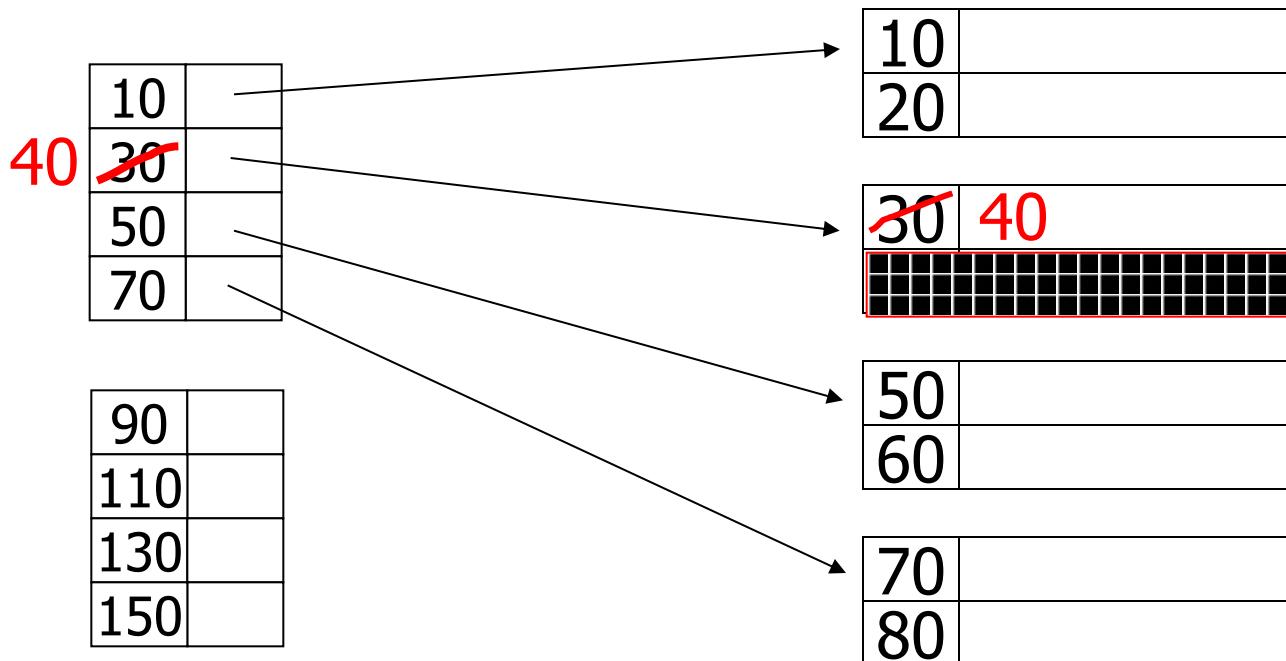
Deletion from sparse index

– delete record 30



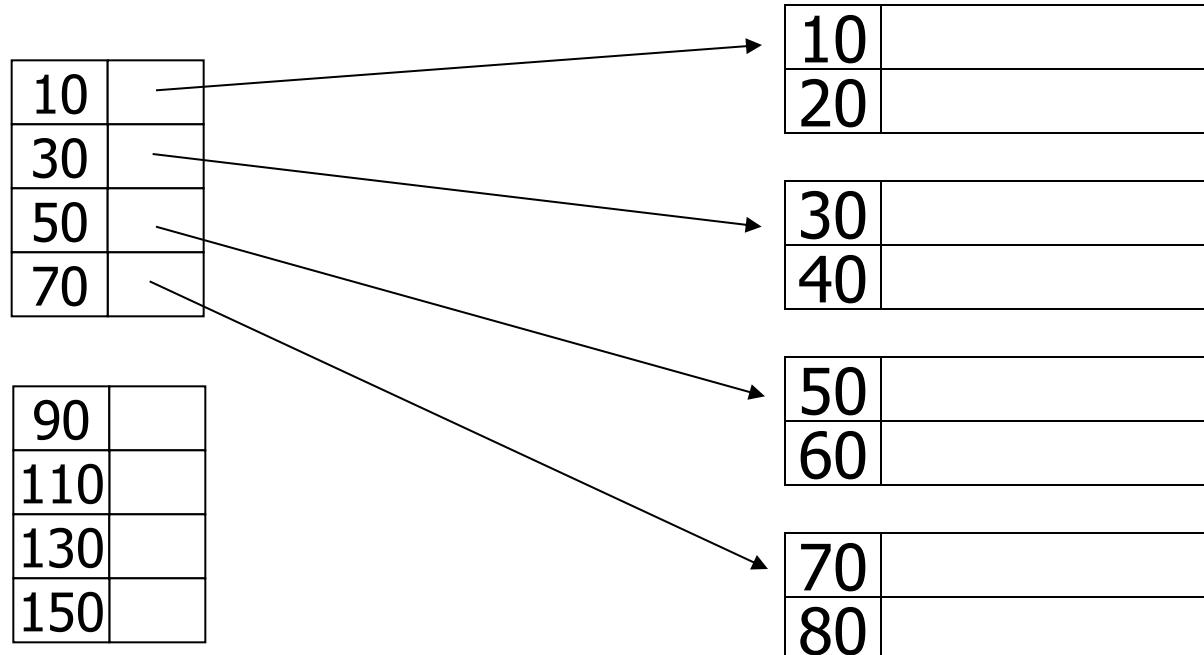
Deletion from sparse index

– delete record 30



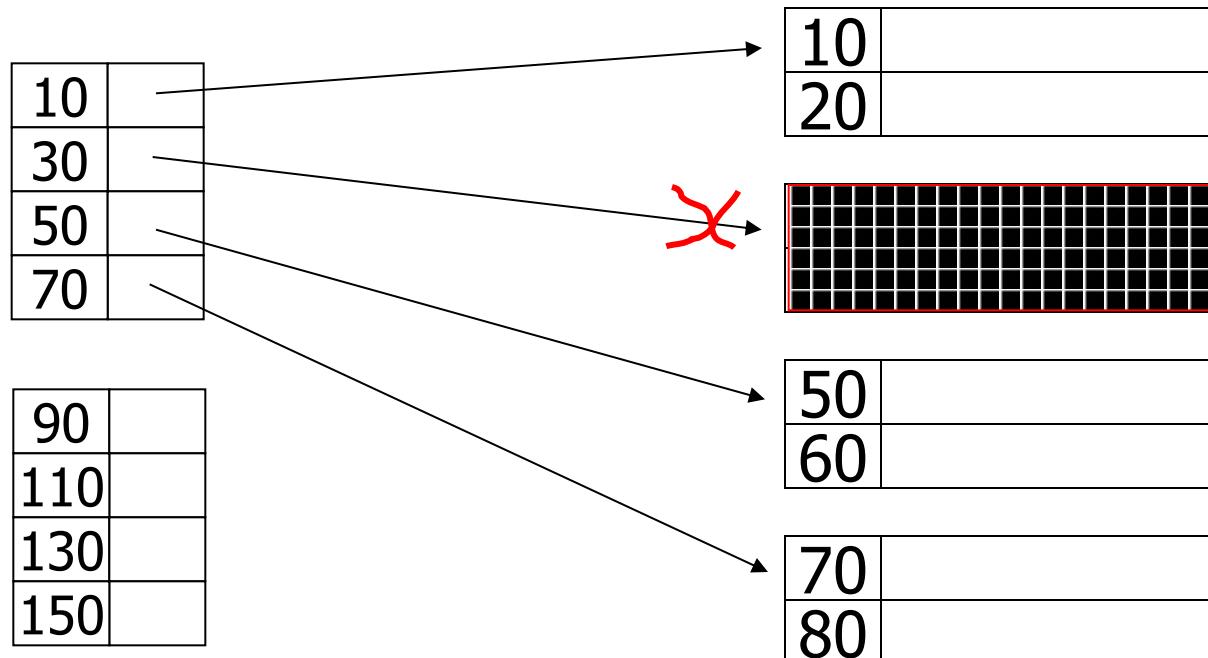
Deletion from sparse index

– delete records 30 & 40



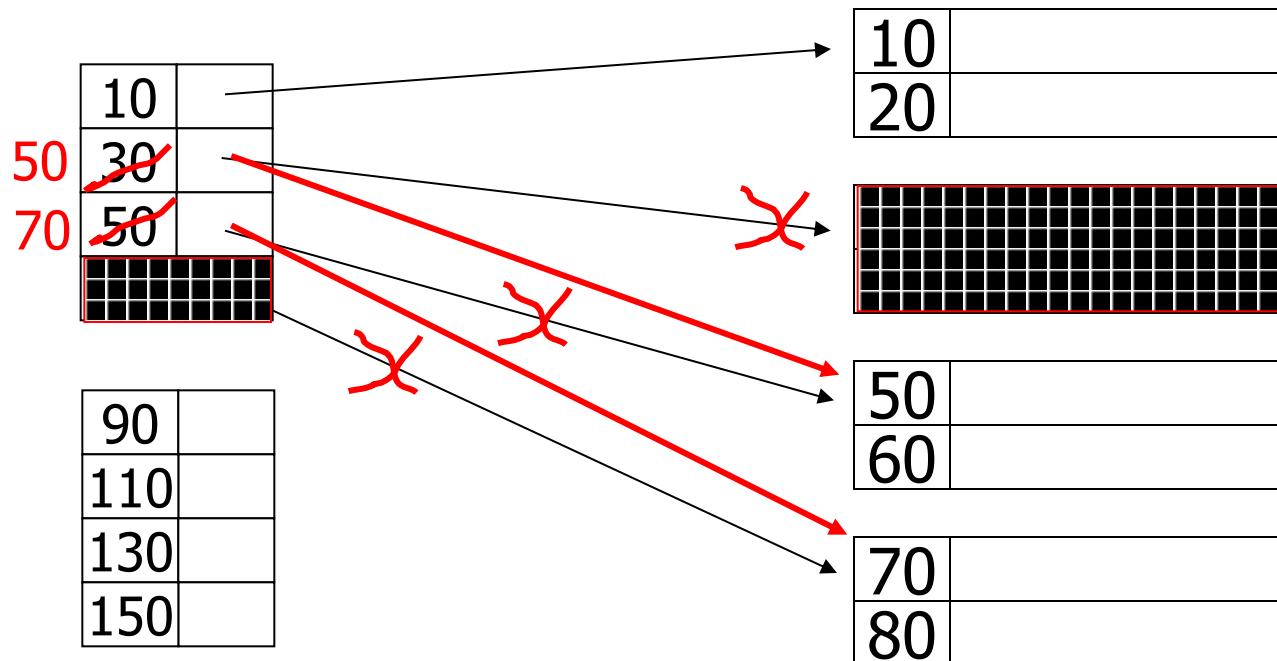
Deletion from sparse index

– delete records 30 & 40

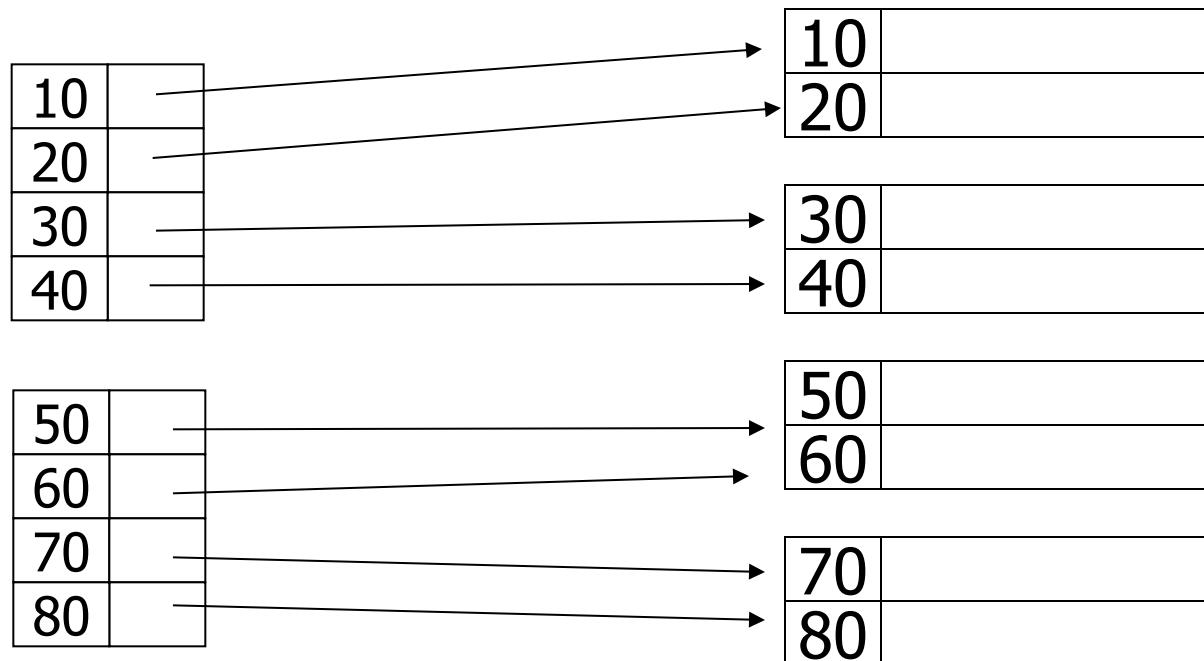


Deletion from sparse index

– delete records 30 & 40

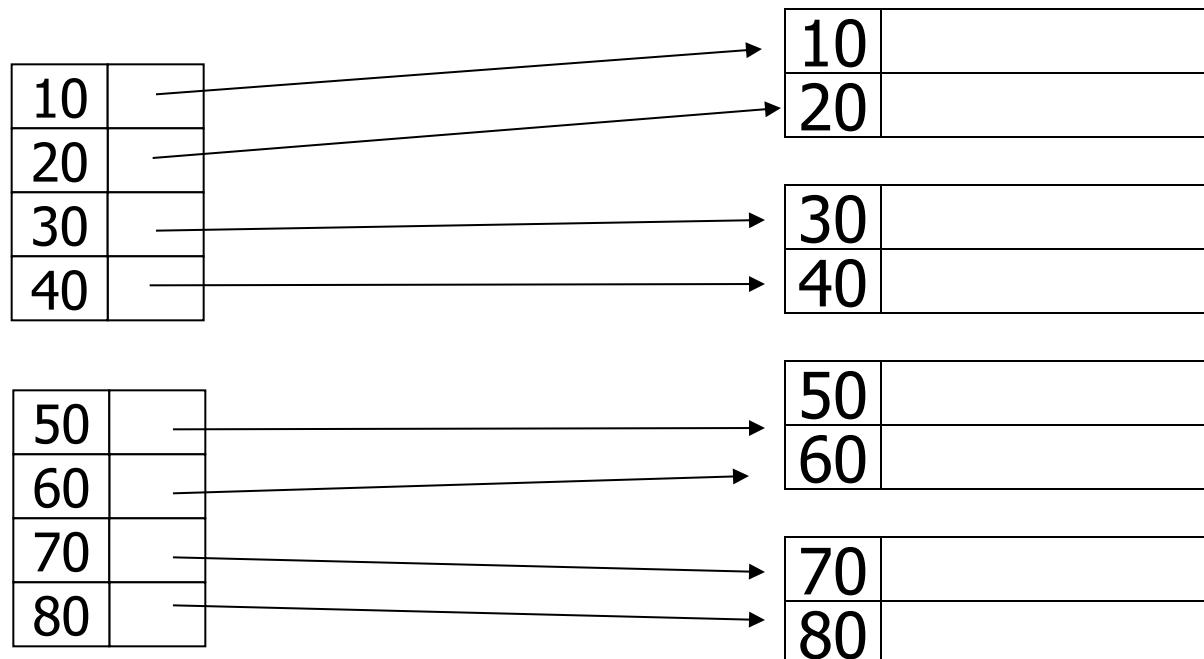


Deletion from dense index



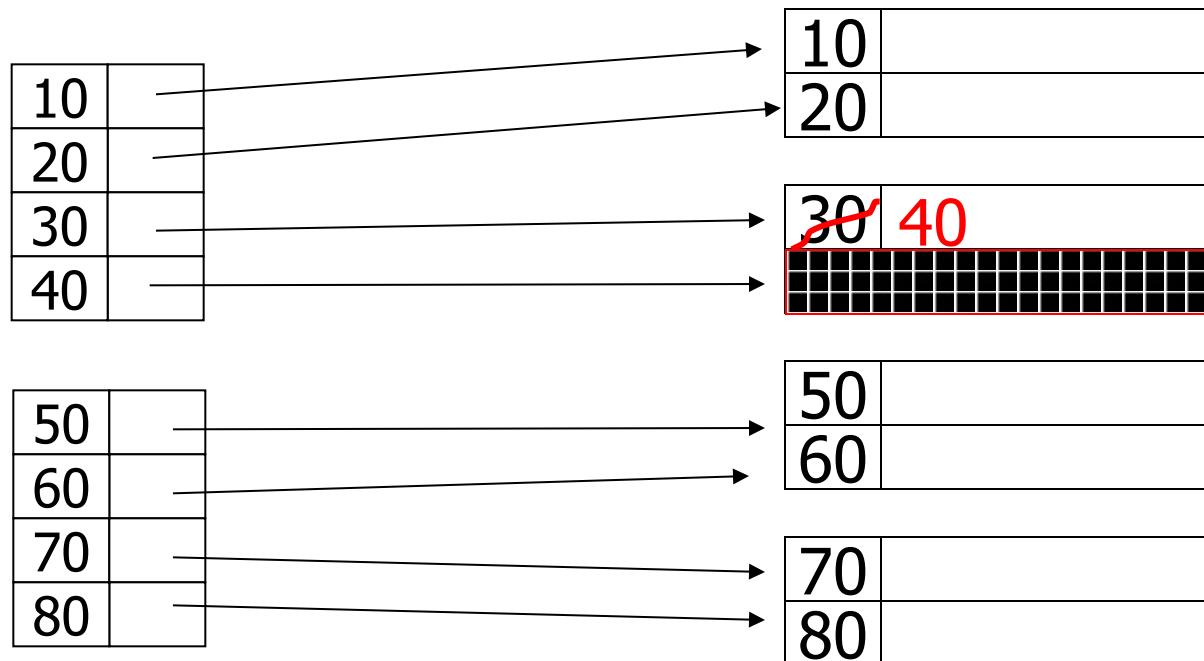
Deletion from dense index

– delete record 30



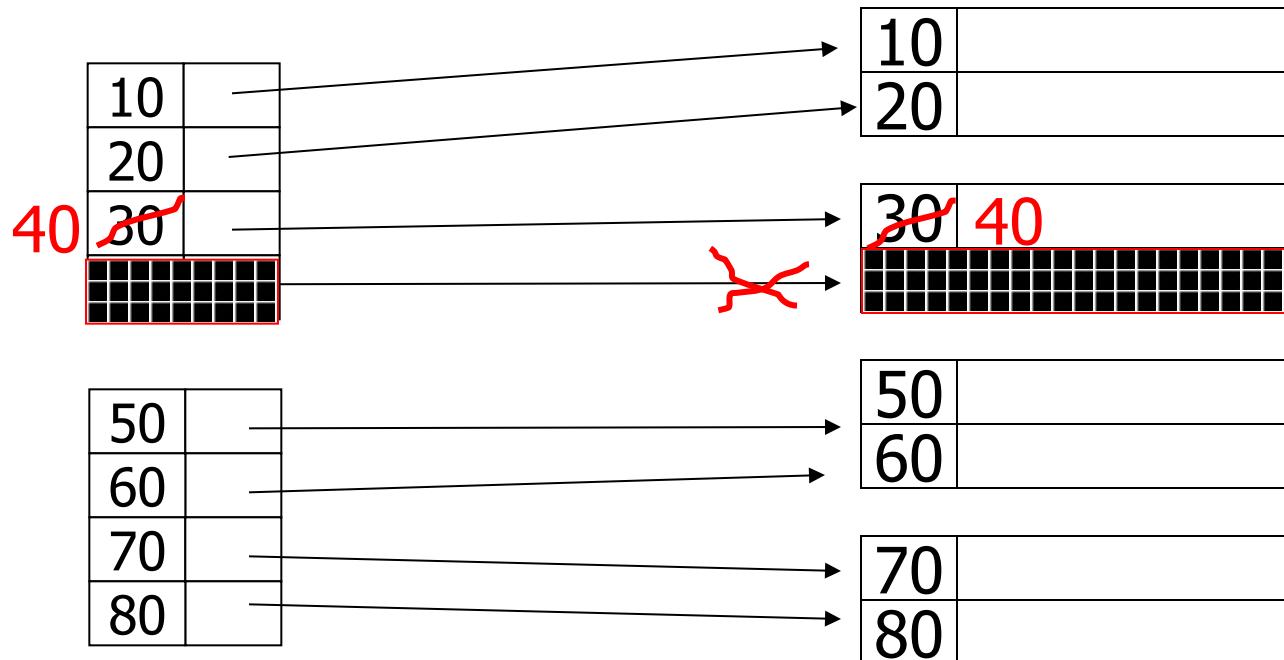
Deletion from dense index

– delete record 30

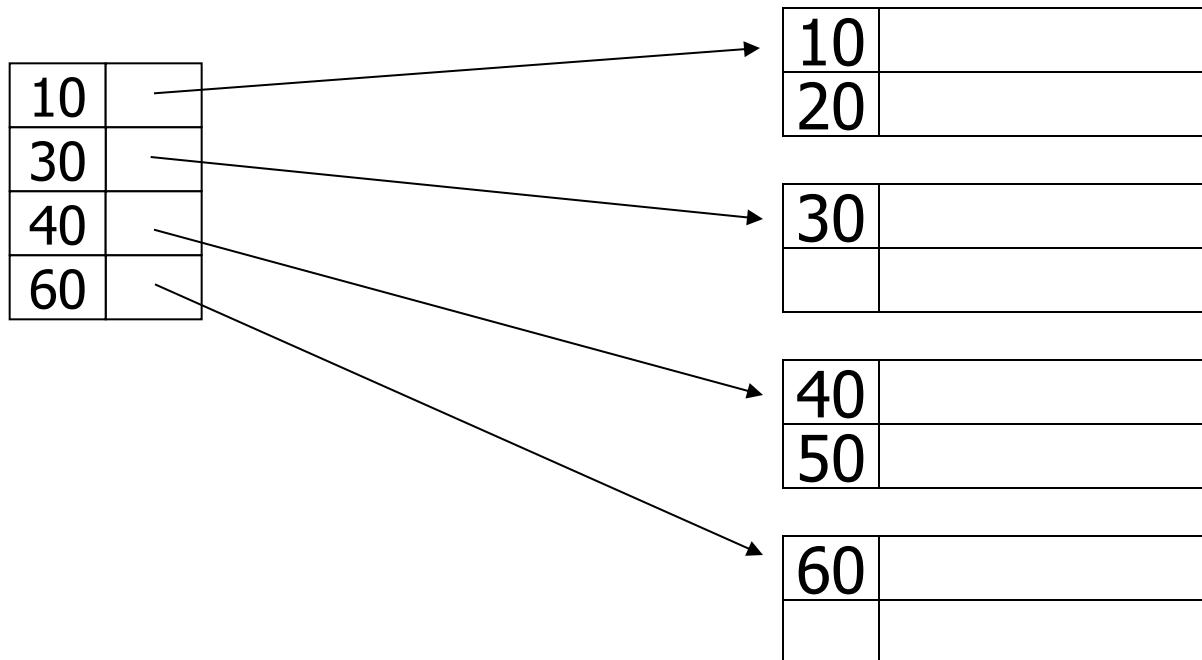


Deletion from dense index

– delete record 30

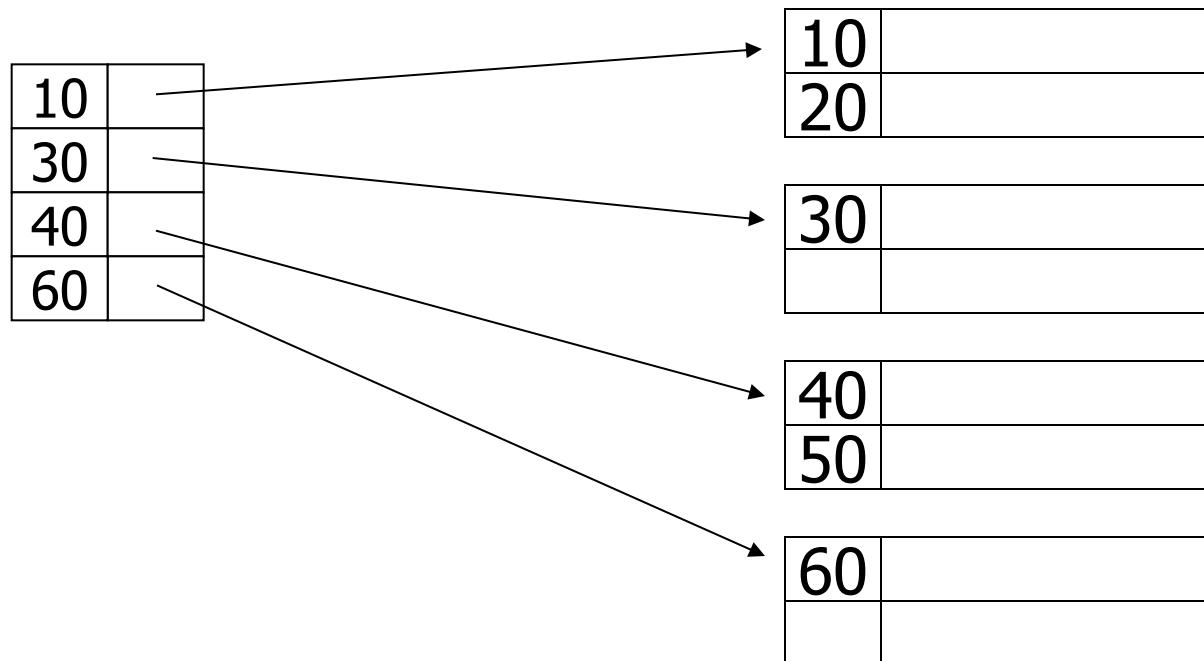


Insertion, sparse index case



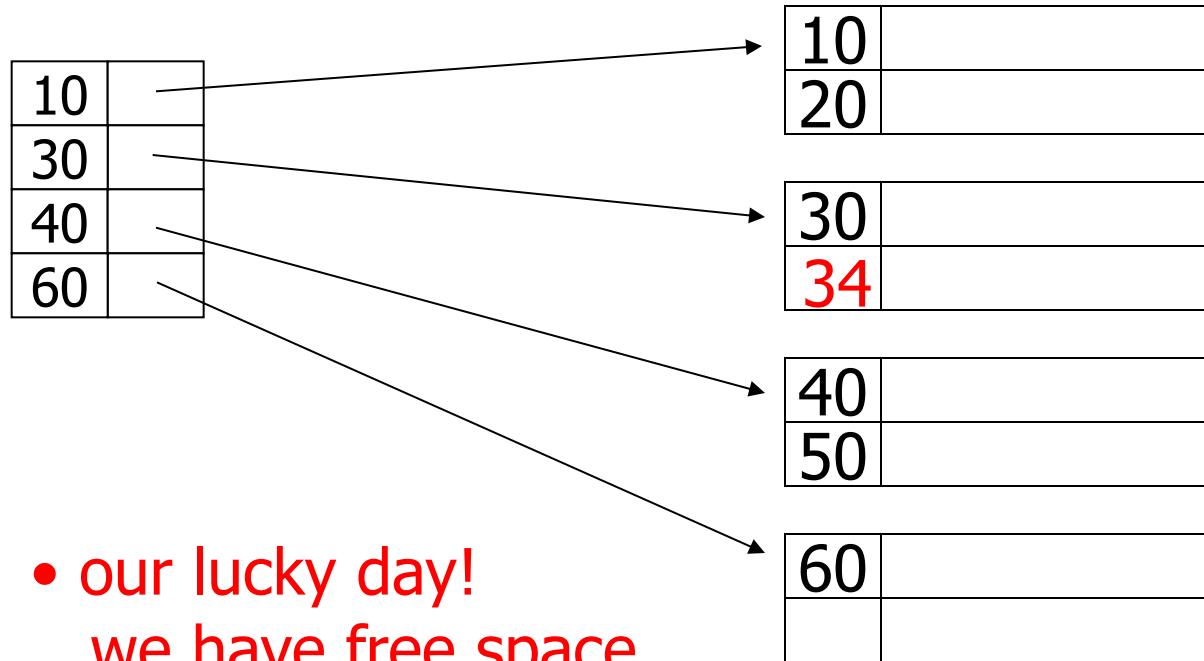
Insertion, sparse index case

– insert record 34



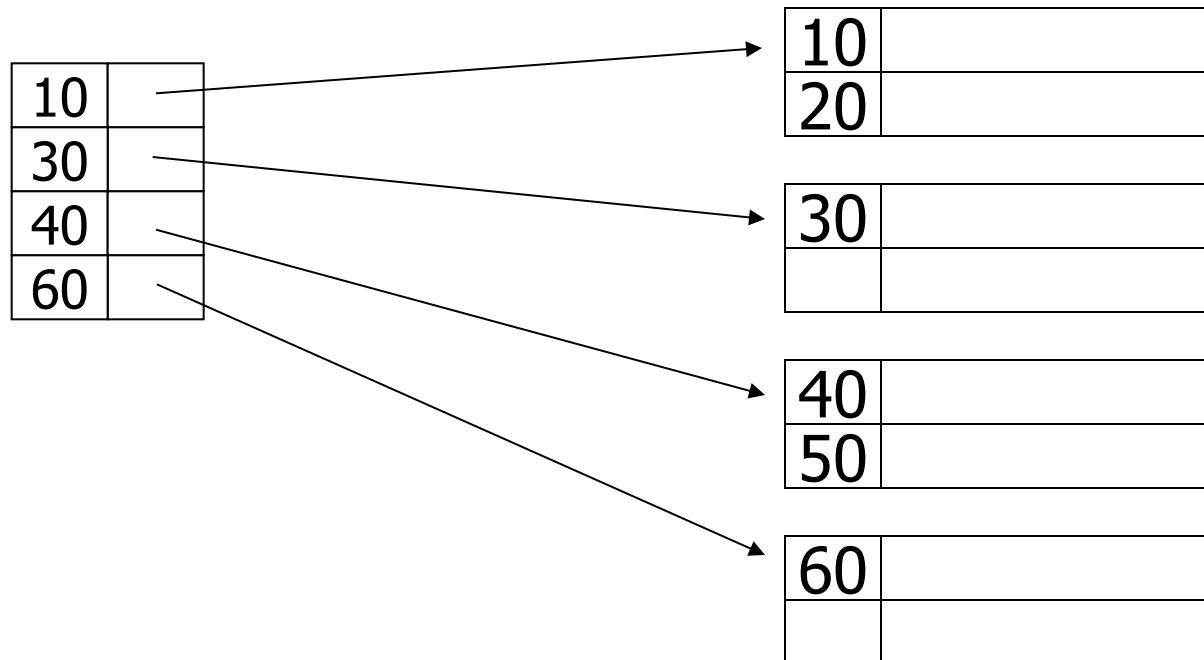
Insertion, sparse index case

– insert record 34



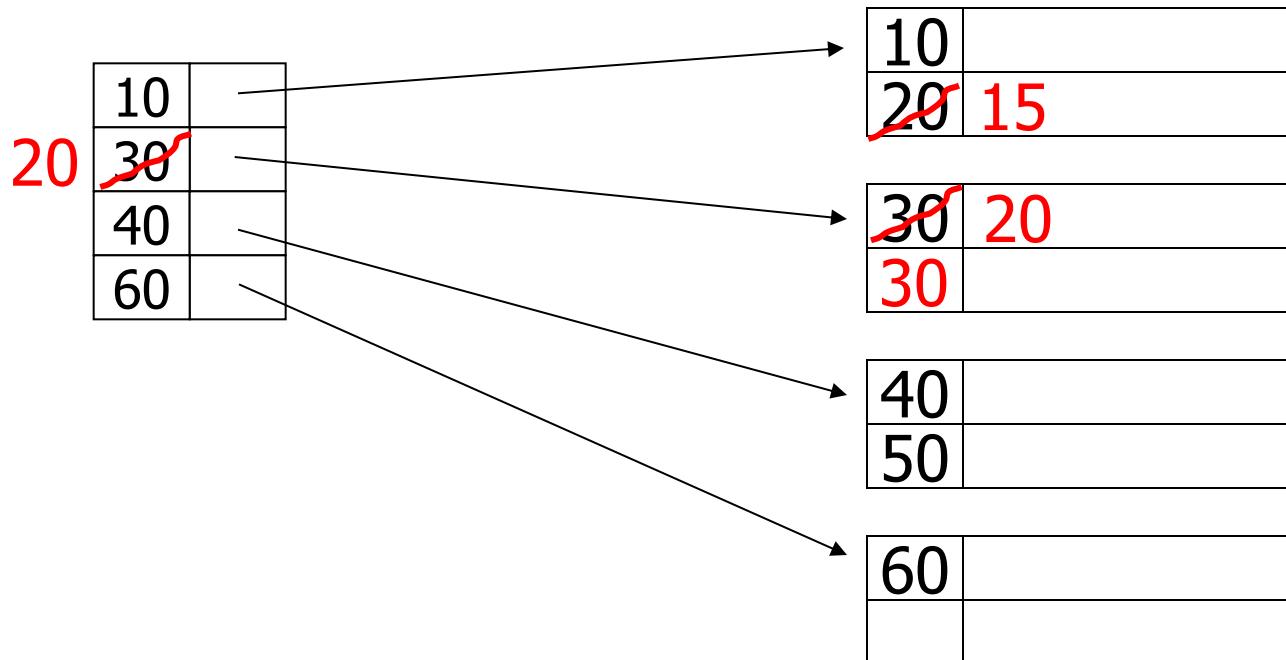
Insertion, sparse index case

– insert record 15



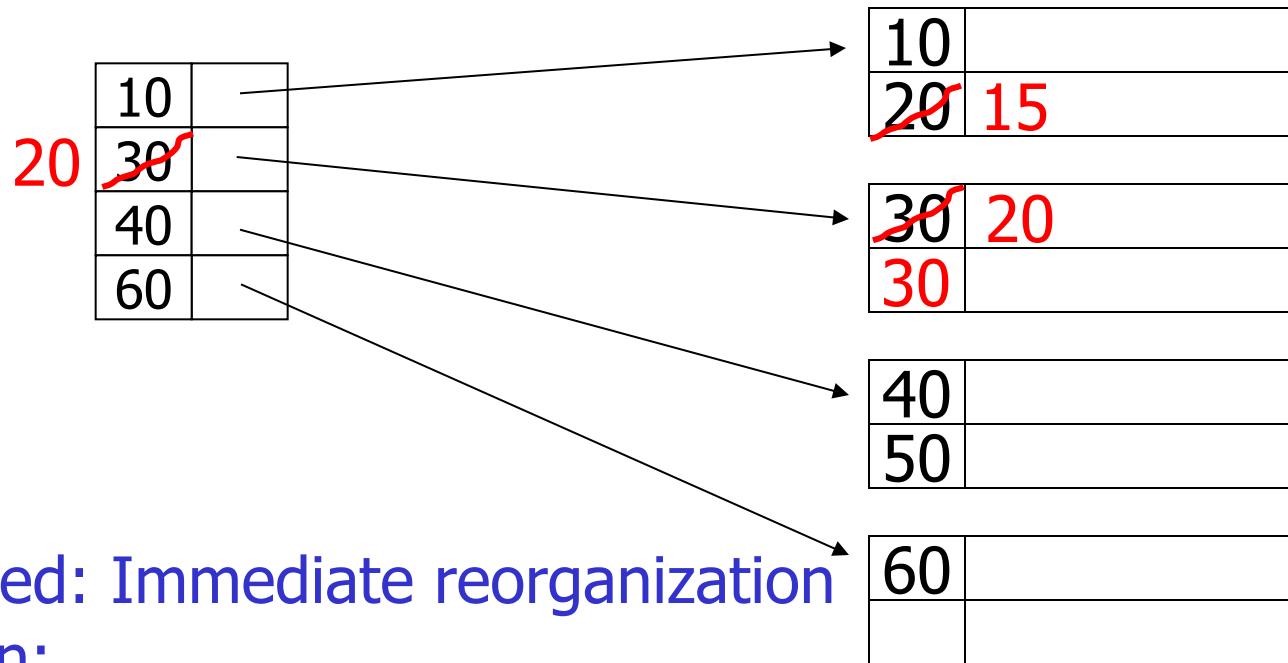
Insertion, sparse index case

– insert record 15



Insertion, sparse index case

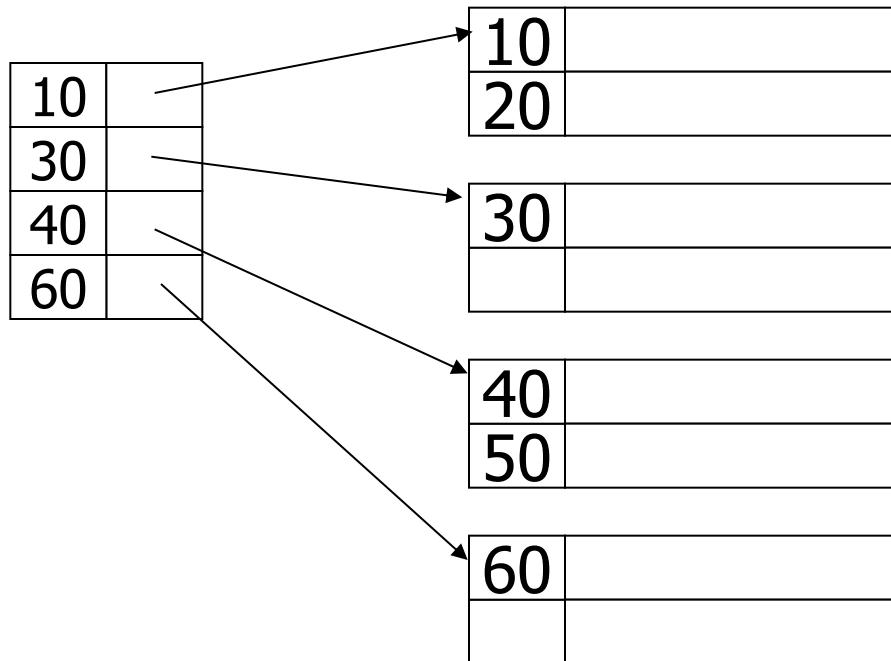
– insert record 15



- Illustrated: Immediate reorganization
- Variation:
 - insert new block (chained file)
 - update index

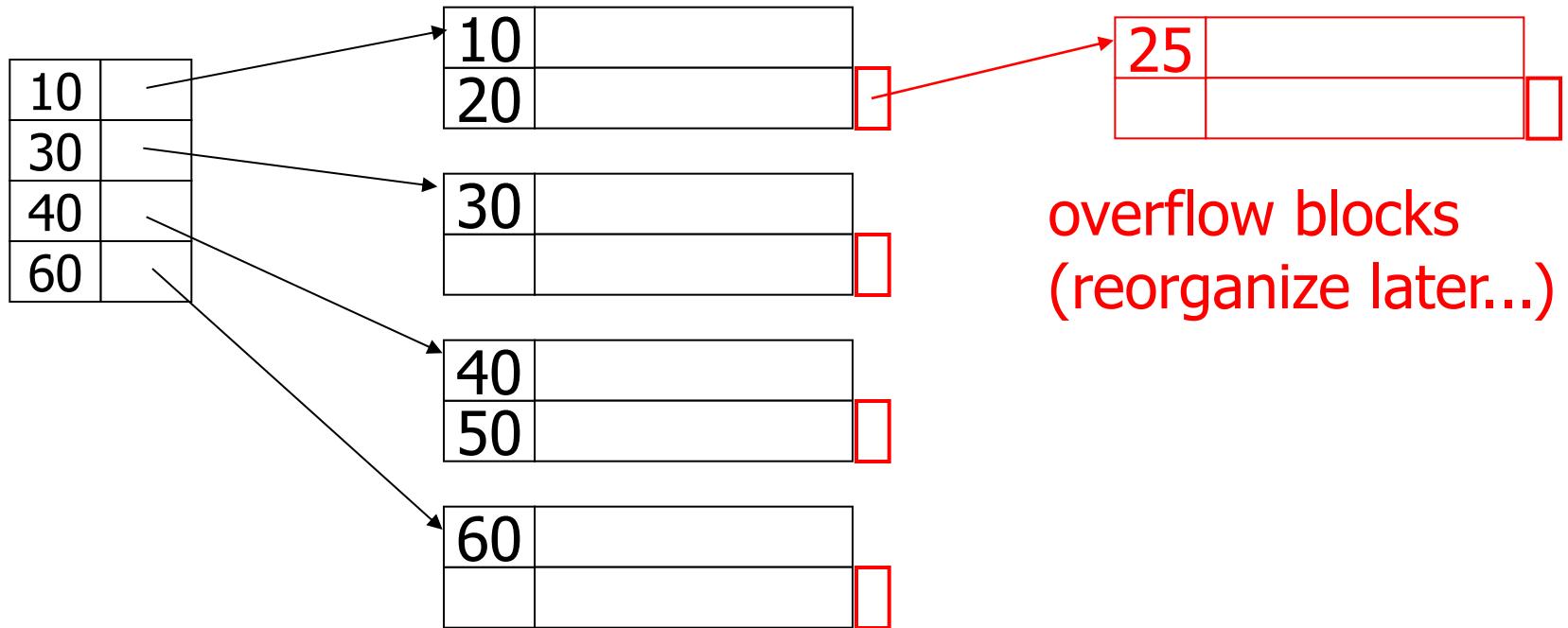
Insertion, sparse index case

– insert record 25



Insertion, sparse index case

– insert record 25



Insertion, dense index case

- Similar
- Often more expensive . . .

Secondary indexes

Sequence
field

30	
50	

20	
70	

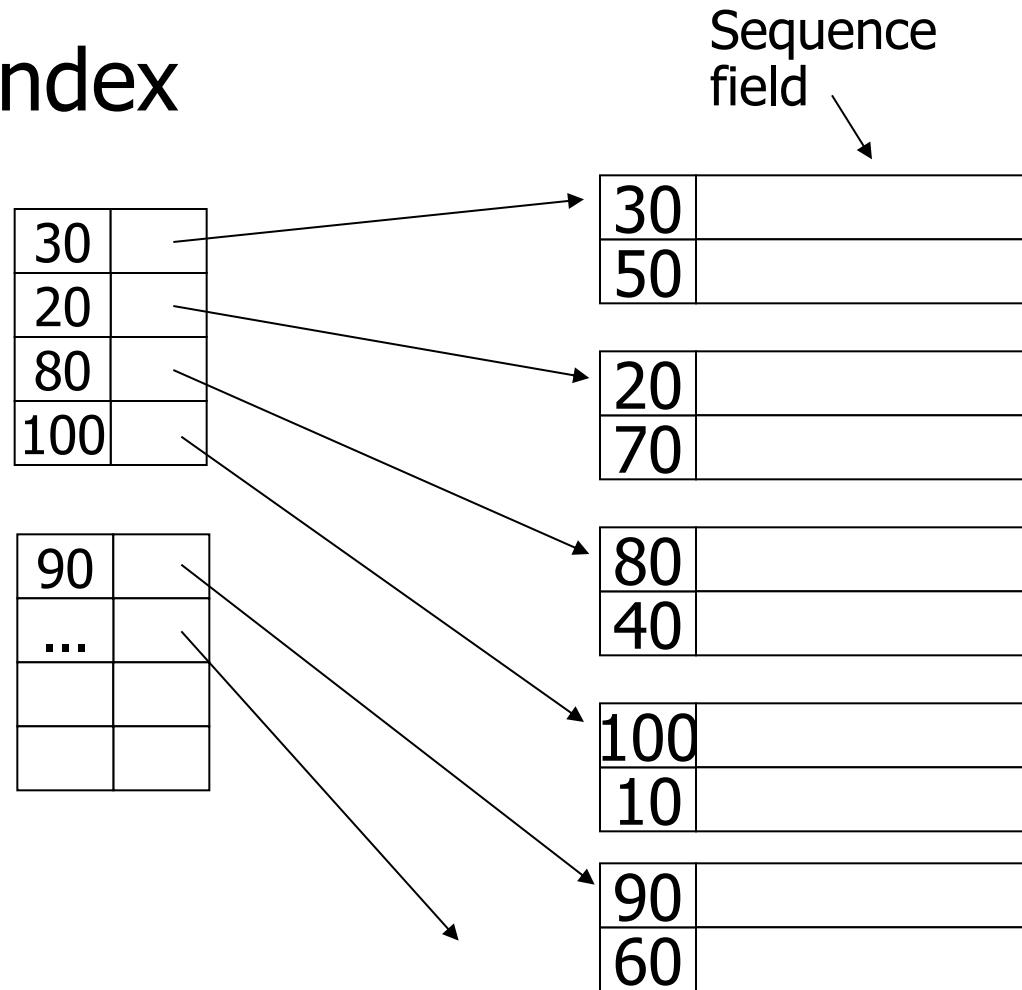
80	
40	

100	
10	

90	
60	

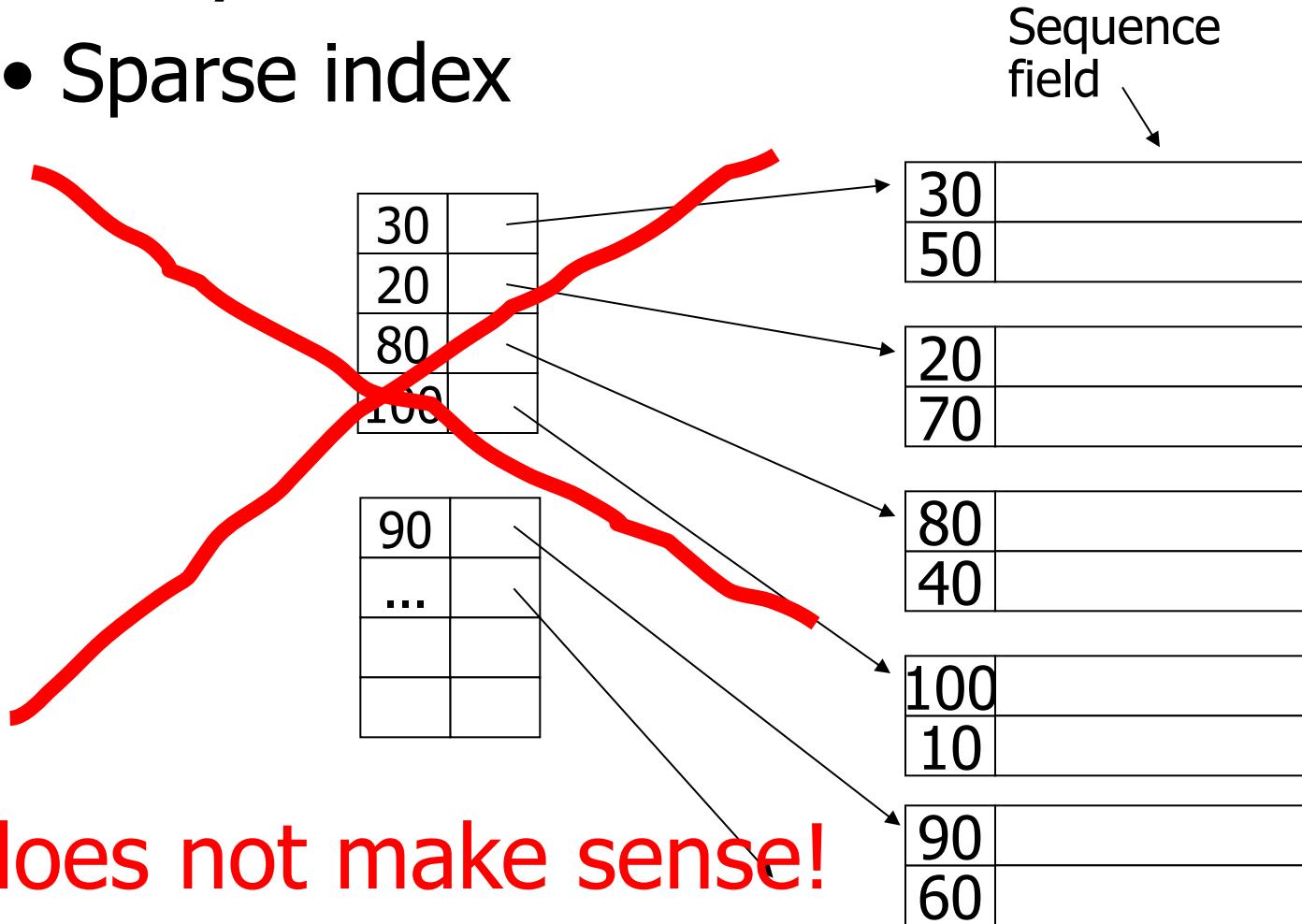
Secondary indexes

- Sparse index



Secondary indexes

- Sparse index



Secondary indexes

- Dense index

Sequence
field

30	
50	

20	
70	

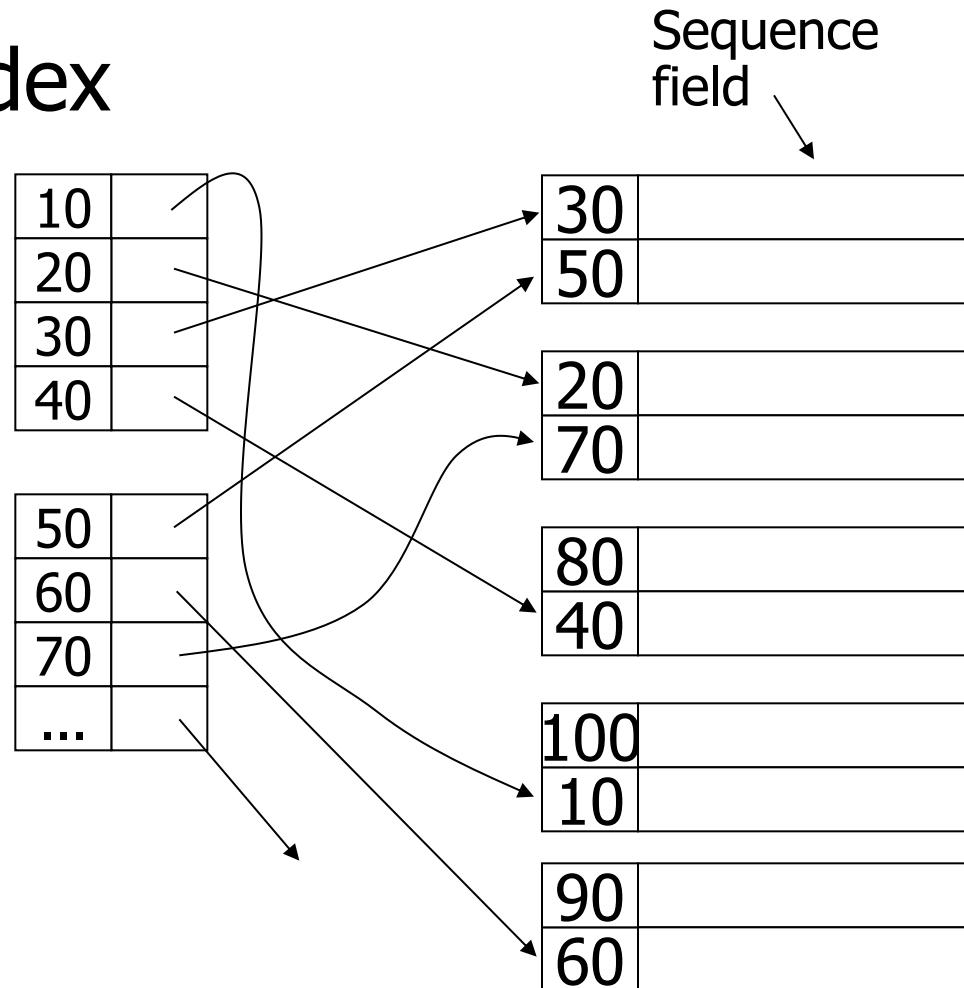
80	
40	

100	
10	

90	
60	

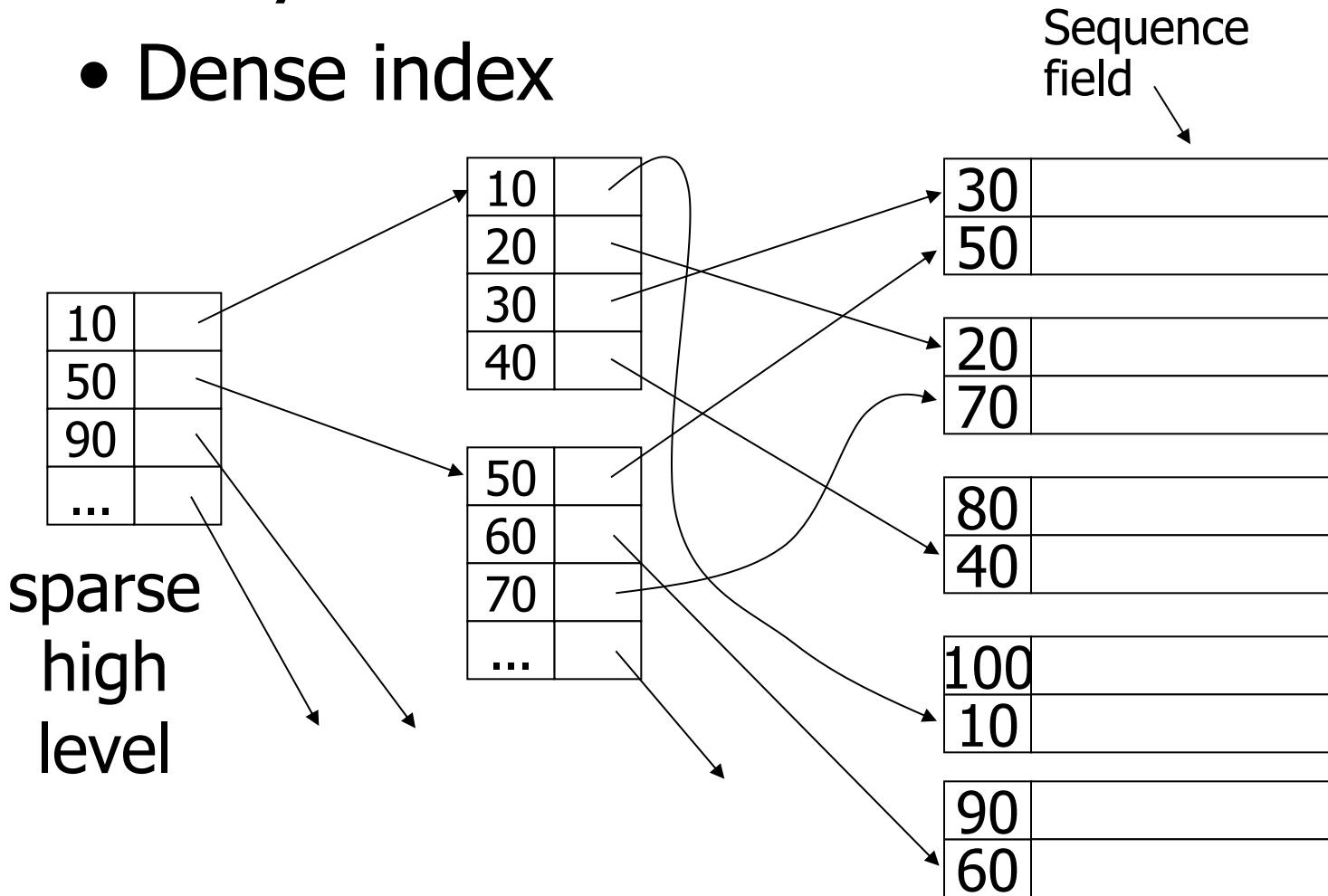
Secondary indexes

- Dense index



Secondary indexes

- Dense index



With secondary indexes:

- Lowest level is dense
- Other levels are sparse

Also: Pointers are record pointers
(not block pointers; not computed)

Conventional indexes

Advantage:

- Simple
- Index is sequential file
good for scans

Disadvantage:

- Inserts expensive, and/or
- Lose sequentiality & balance

Hashing

Dirk Van Gucht

Based on slides by Hector Garcia-Molina

Hashing: hash function

- Let K be a domain of key values (K can be very large)
- Let $R = [0, m)$ (usually $m \ll |K|$) be a range of values
- A **hash function** h maps K to R

$$h: K \rightarrow R$$

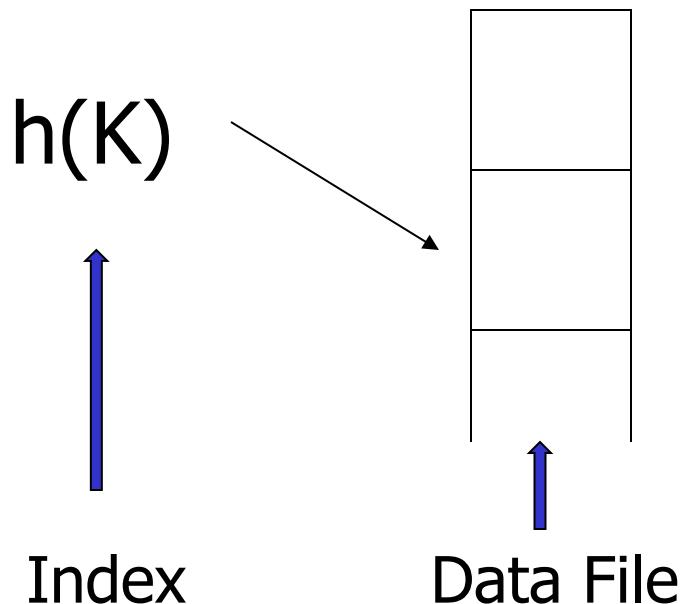
Example: $h(k) = k \bmod m$

- Given a record r with key value k , $h(k)$ provides an address (name) of a **bucket** in which to store r
- A bucket is stored in secondary memory as a block or a list of blocks
- Retrieving a bucket (and the records therein) can be done in $O(n)$ where n is the number of blocks that store the bucket.

Hash function: collision

- Let r_1 and r_2 be two records with key values k_1 and k_2
- We permit that $k_1 = k_2$
- We say that h has a **collision for r_1 and r_2** if $h(k_1) = h(k_2)$
Consequence: r_1 and r_2 will be stored in the same bucket
- If K is the domain of a primary key, a collision will store records with different key values in same bucket (not desirable)
- If K is not the domain of a primary key, different records with the same key values will be placed in the same bucket (desirable). **Partitioning.**
- The latter property is exploited in key-value stores since records with the same key value will be sent to the same reducer

Next: example to illustrate inserts, overflows, deletes



EXAMPLE 2 records/bucket

(For simplicity, we identify a record with its key value)

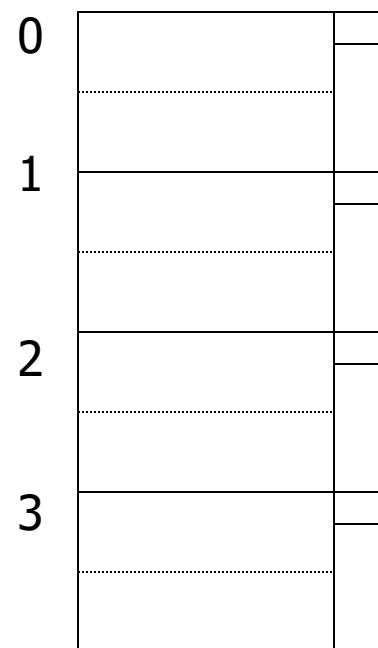
INSERT:

$$h(a) = 1$$

$$h(b) = 2$$

$$h(c) = 1$$

$$h(d) = 0$$



EXAMPLE 2 records/bucket

INSERT:

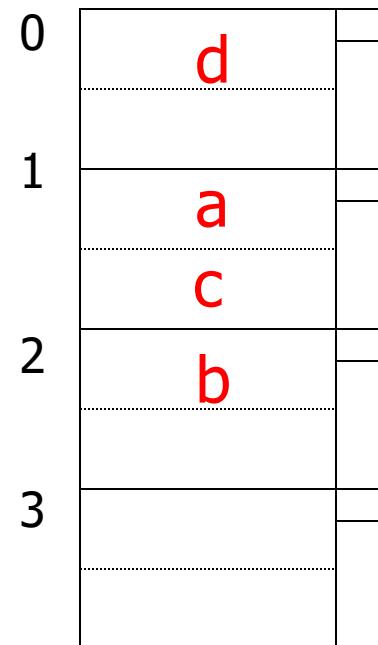
$$h(a) = 1$$

$$h(b) = 2$$

$$h(c) = 1$$

$$h(d) = 0$$

$$h(e) = 1$$



EXAMPLE 2 records/bucket

INSERT:

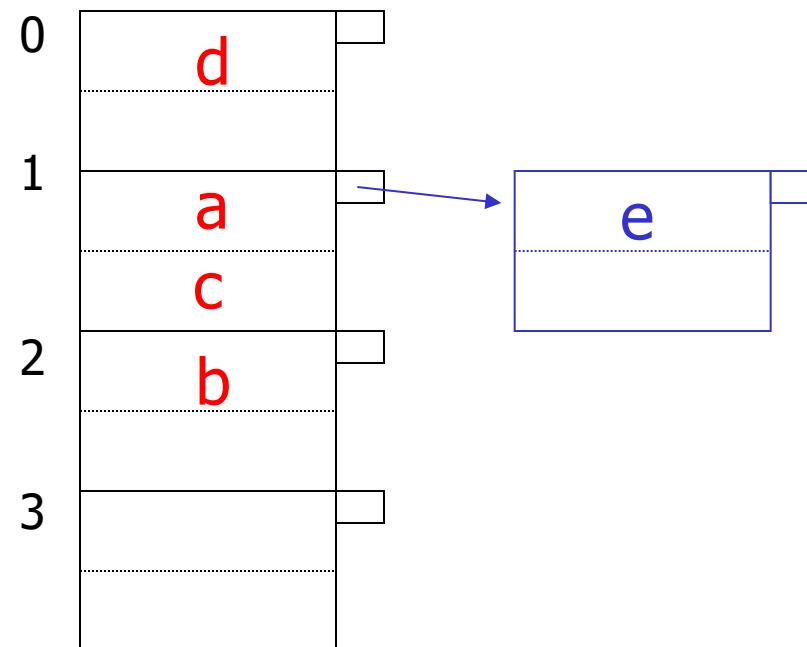
$$h(a) = 1$$

$$h(b) = 2$$

$$h(c) = 1$$

$$h(d) = 0$$

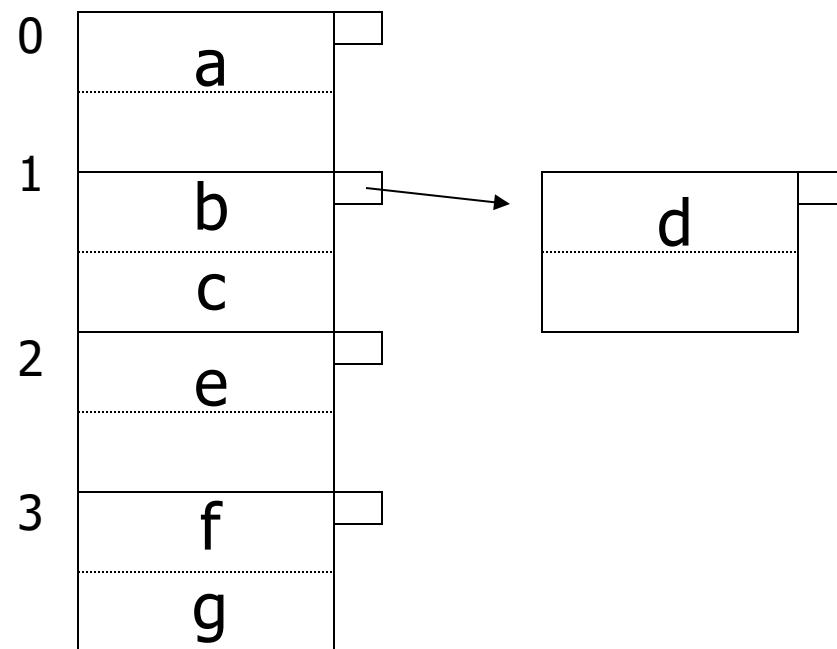
$$h(e) = 1$$



EXAMPLE: deletion

Delete:

e
f



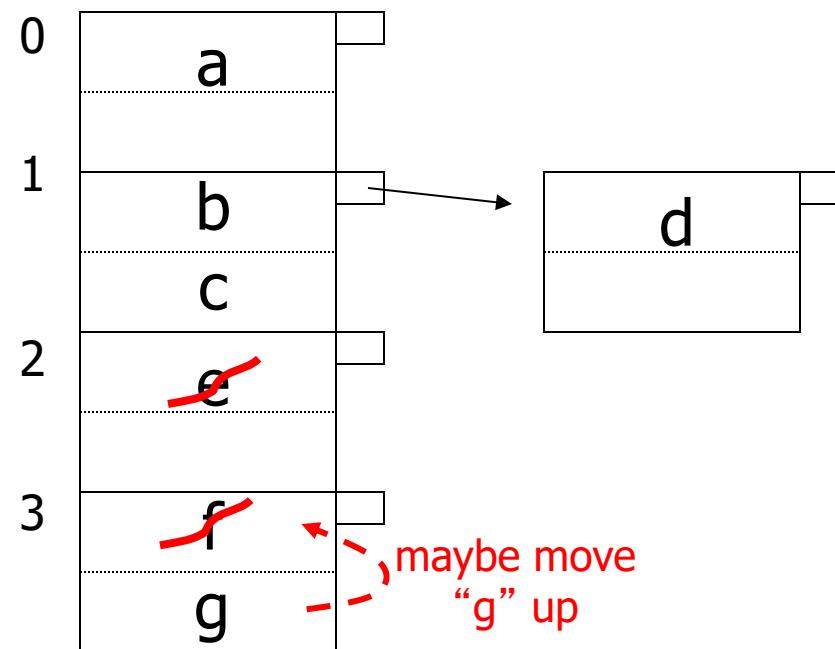
EXAMPLE: deletion

Delete:

e

f

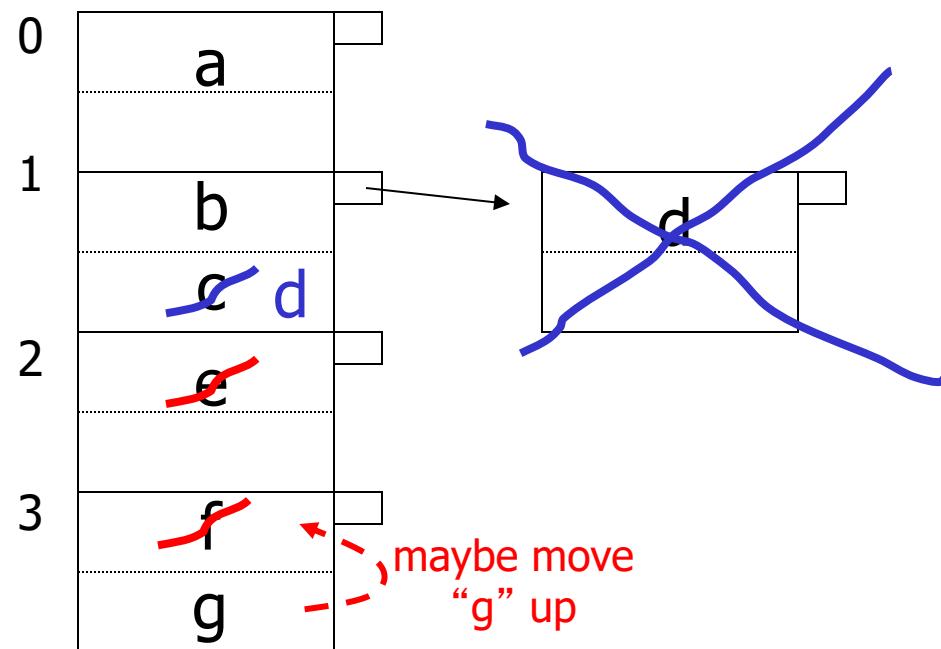
C



EXAMPLE: deletion

Delete:

e
f
C

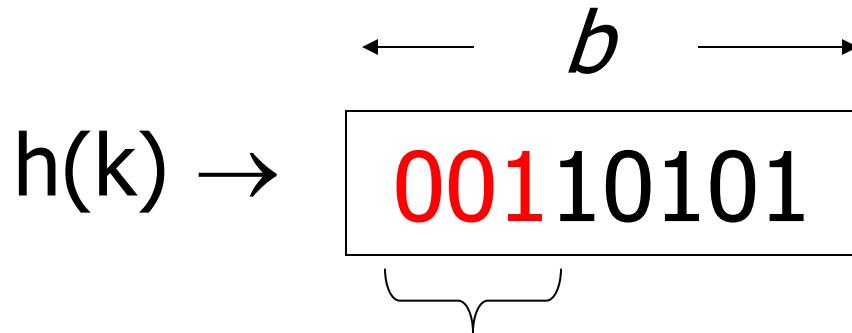


How do we cope with growth?

- Overflows and reorganizations
 - Reorganization can be done by enlarging the range R and changing the hash function
 - Reorganization requires complete rehashing and is linear in the $|Data\ file|$
- Dynamic hashing (extensible hashing)

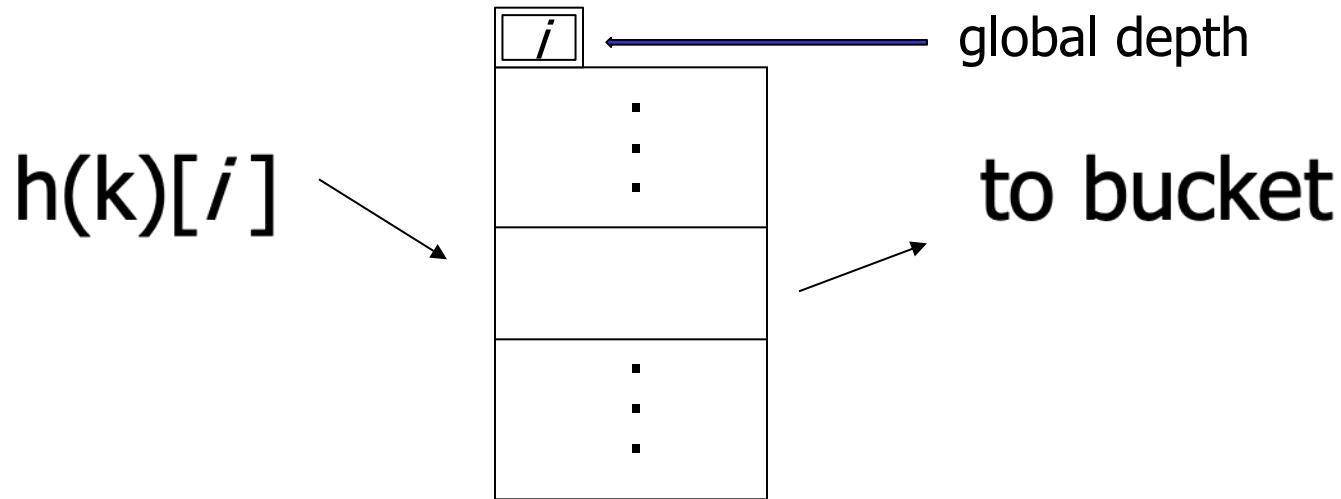
Extensible hashing: two ideas

(a) Use i of b bits output by hash function



use $i \rightarrow$ grows/shrinks over time....

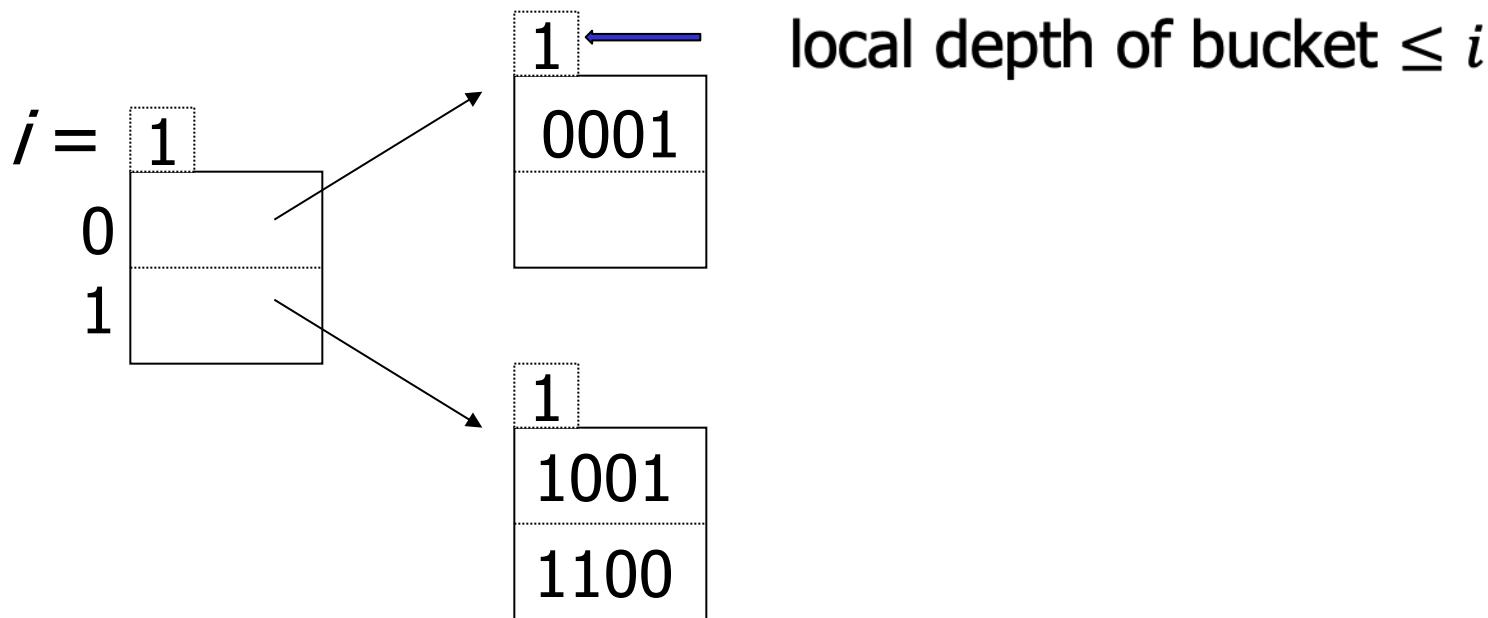
(b) Use directory of size 2^i



$h(k)$ has b bits, but we will only look at its first i bits

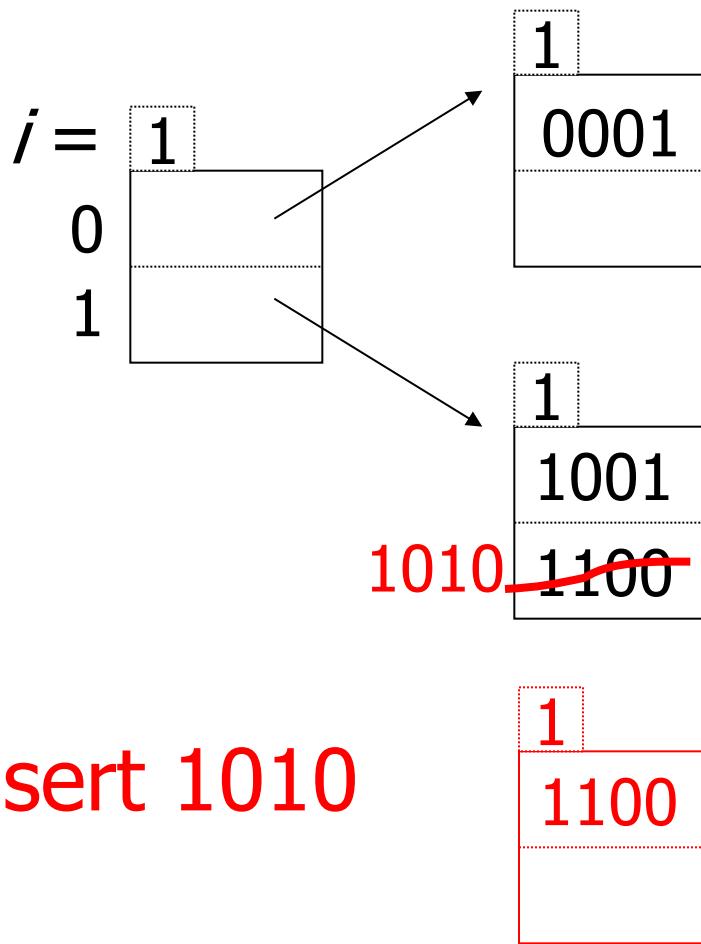
$h(k)[i]$ consists of the first i bits of $h(k)$
these i bits specify the position for k the
directory

Example: $h(k)$ is 4 bits; 2 keys/bucket



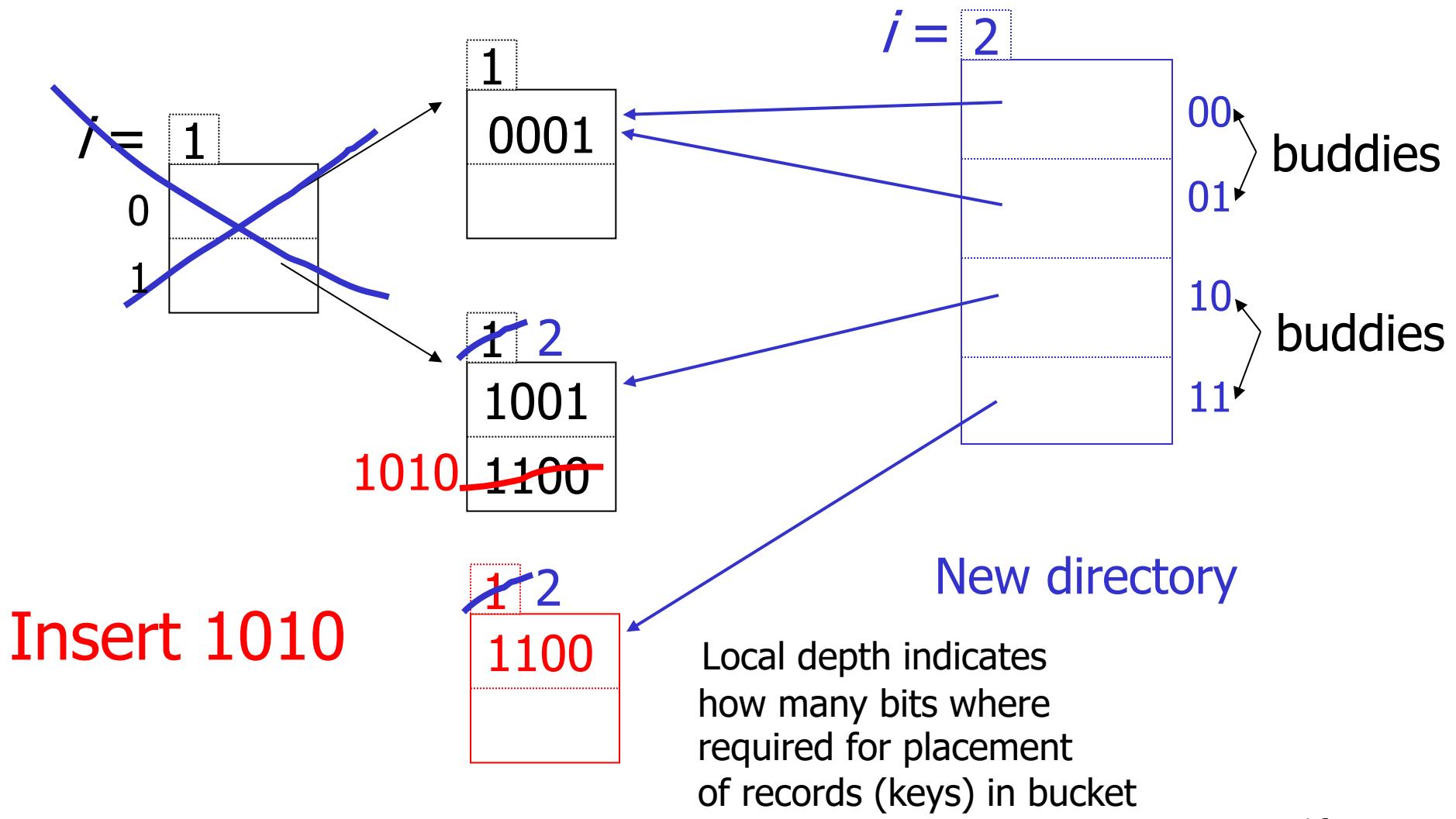
Insert 1010

Example: $h(k)$ is 4 bits; 2 keys/bucket

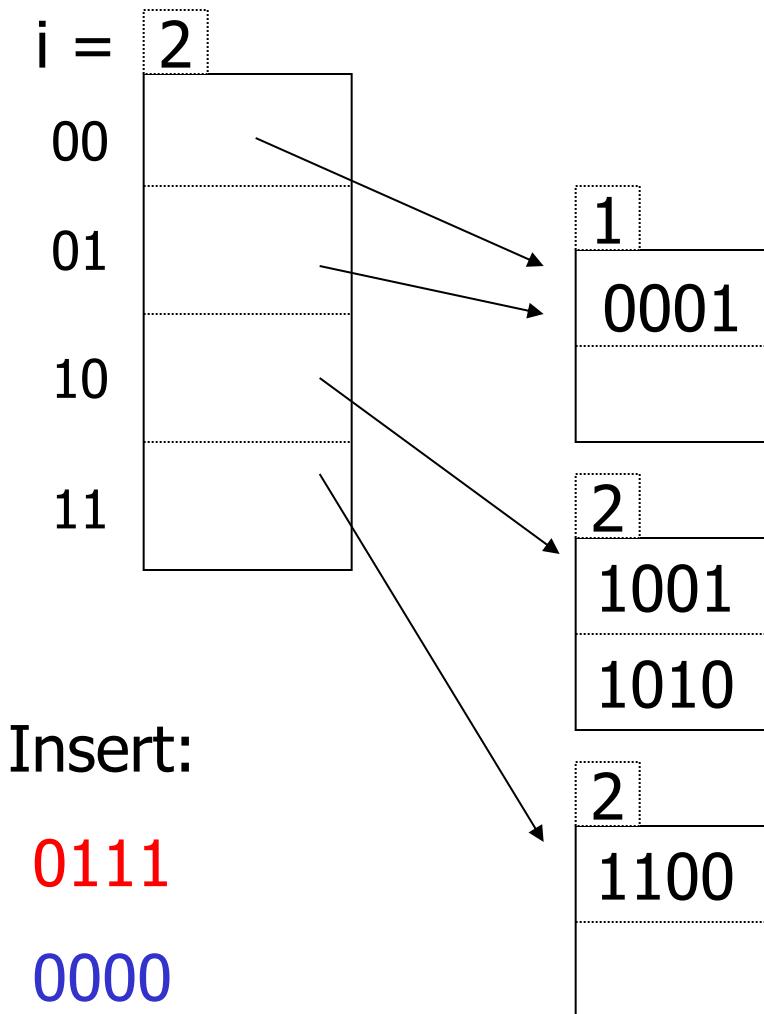


Main idea:
we need 2 bits to
distinguish
1001
1100
1010

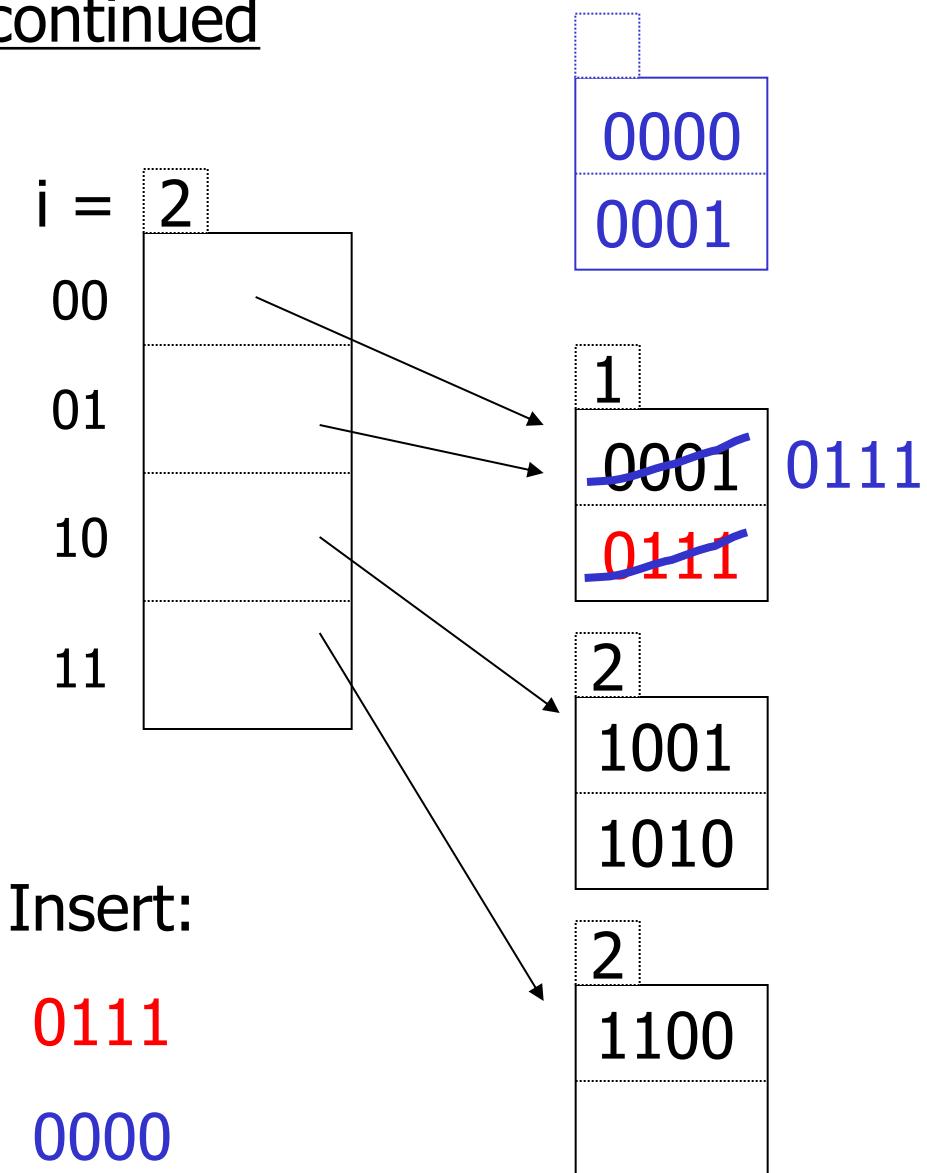
Example: $h(k)$ is 4 bits; 2 keys/bucket



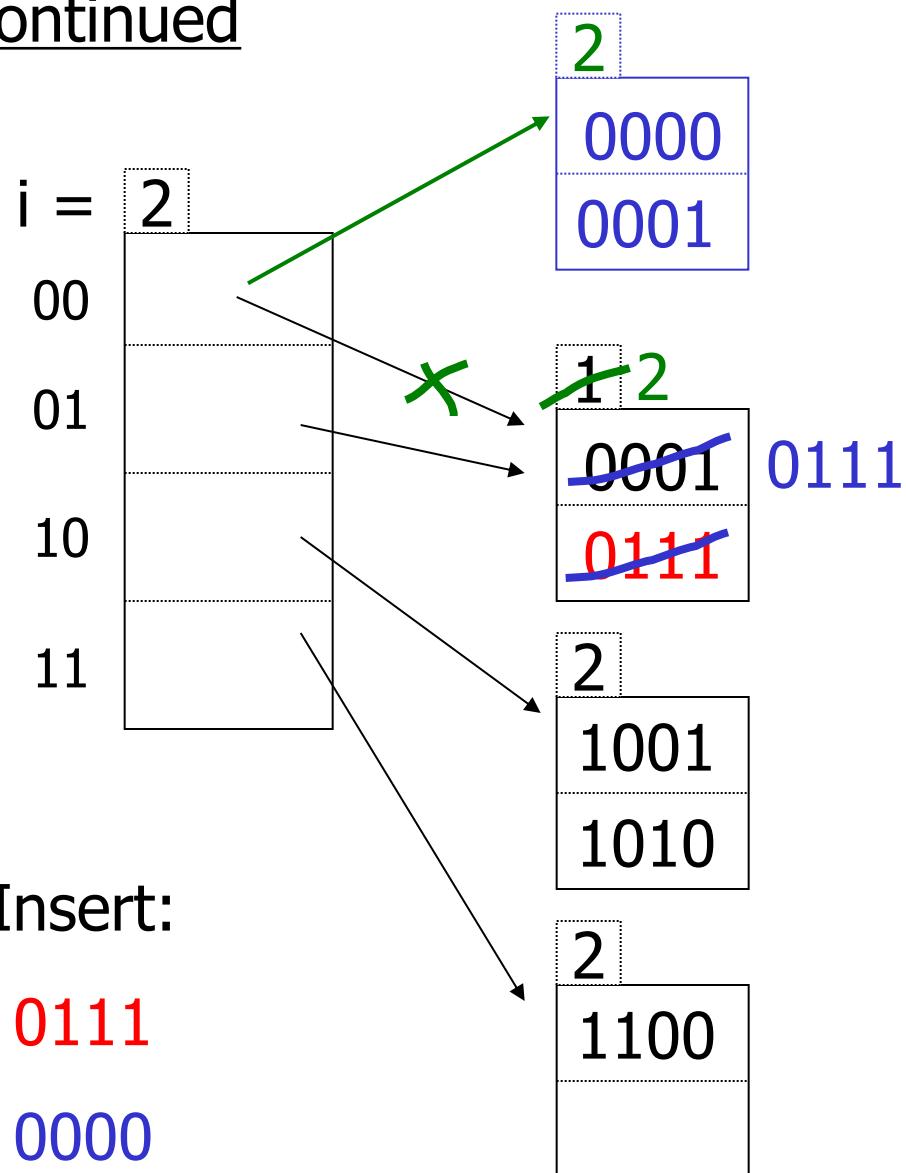
Example continued



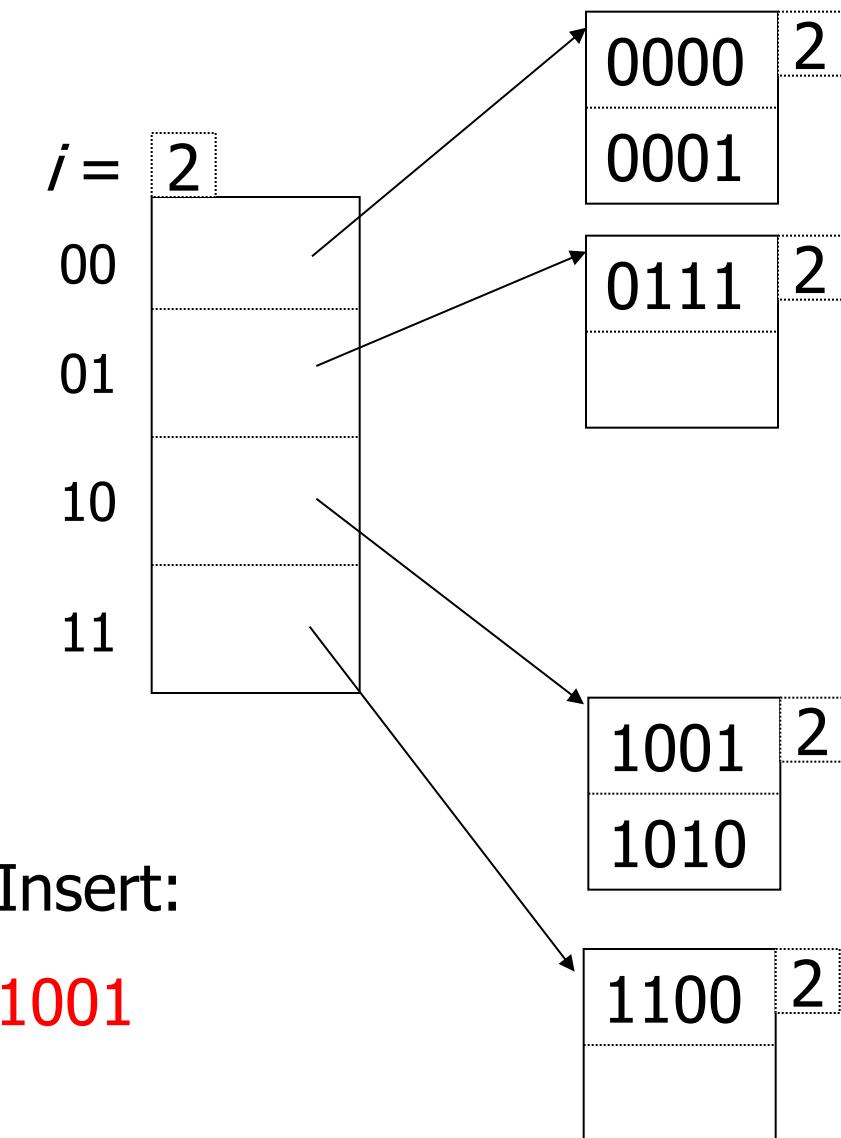
Example continued



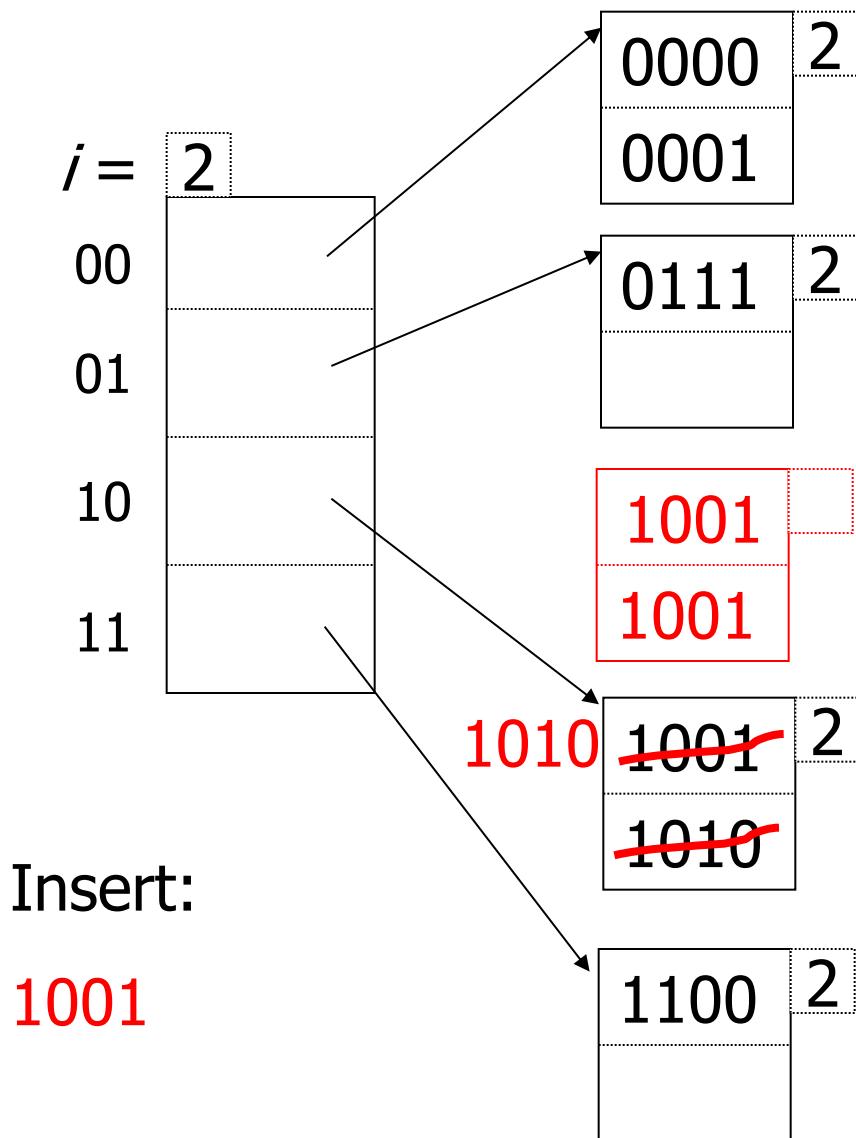
Example continued



Example continued

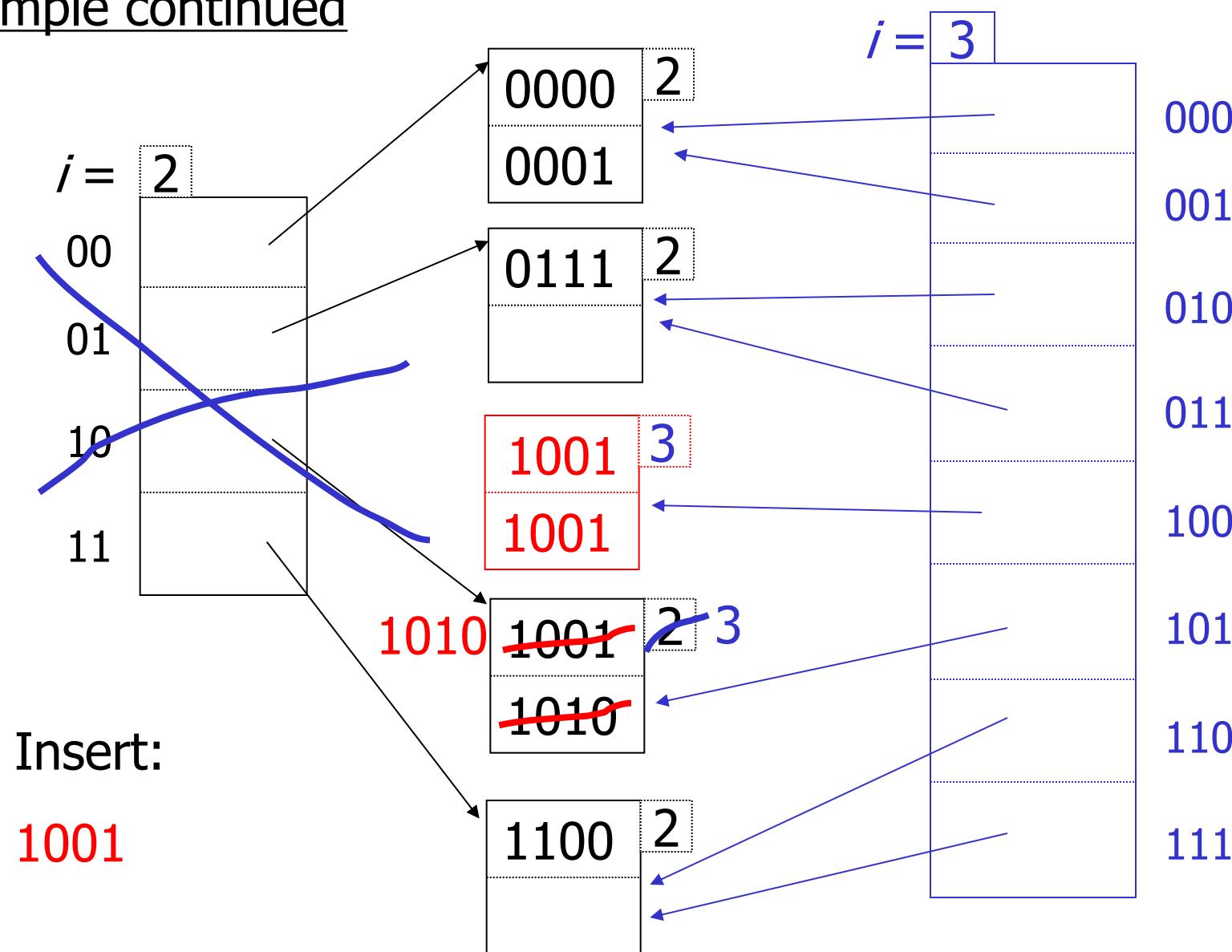


Example continued



21

Example continued



Extensible hashing: deletion

- Merge blocks and cut directory if possible
(Reverse insert procedure)
- Two blocks can be merged after a deletion if all records in these blocks can fit in a single block
- One of these two blocks can be removed and the local depth of the other block one can be decreased by 1
- If **all** local depths are strictly smaller than the global depth i , then the directory can be cut in half and the

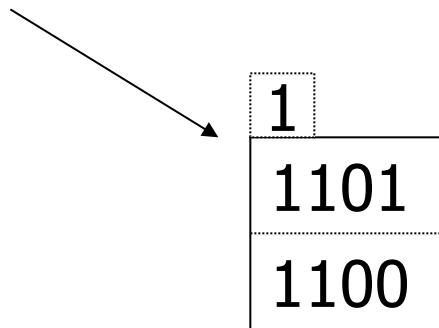
Deletion example:

- Run through insert example in reverse!

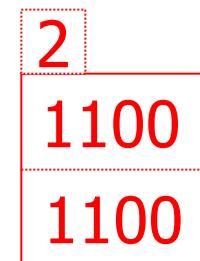
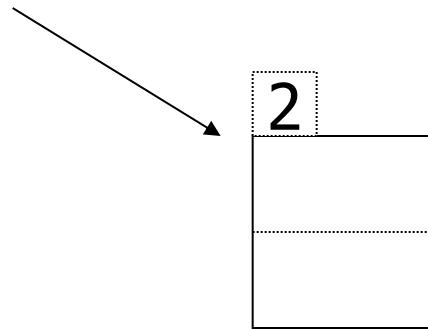
Note: Still need overflow chains

- Example: many records with duplicate keys

insert 1100

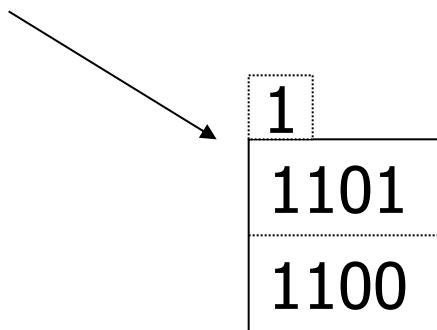


if we split:

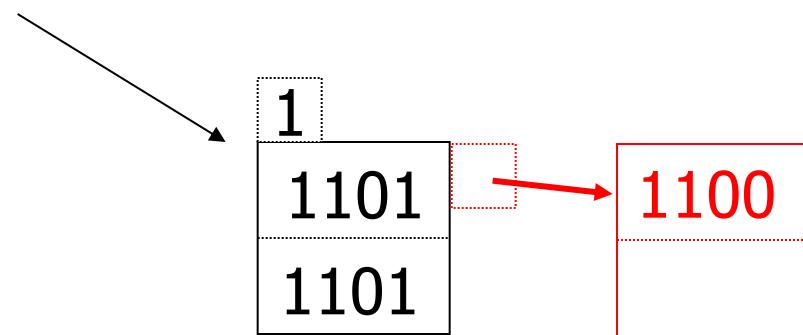


Solution: overflow chains

insert 1100



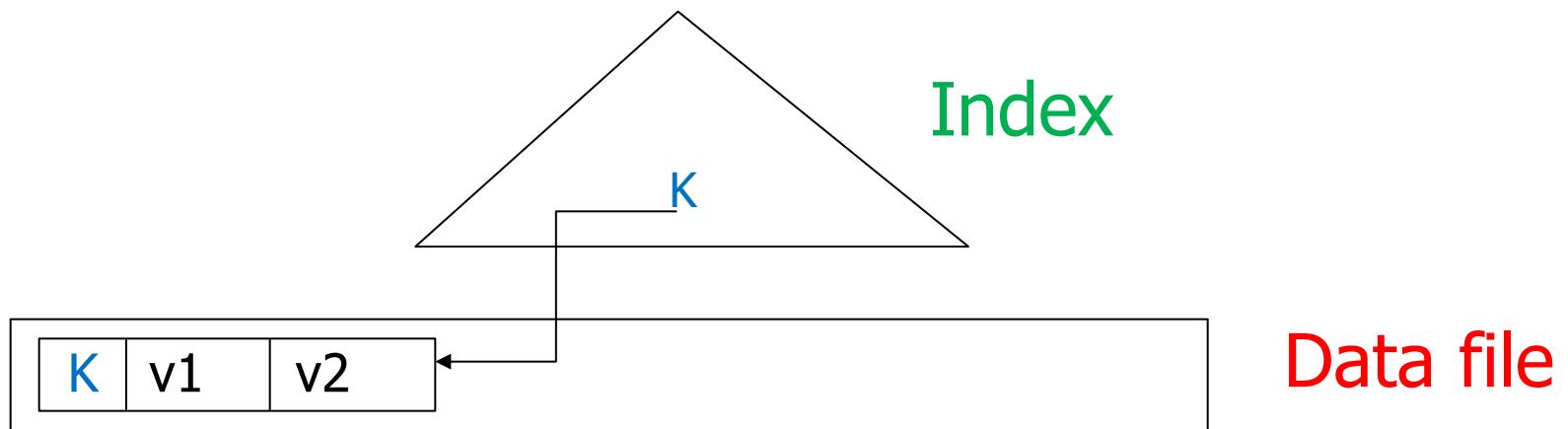
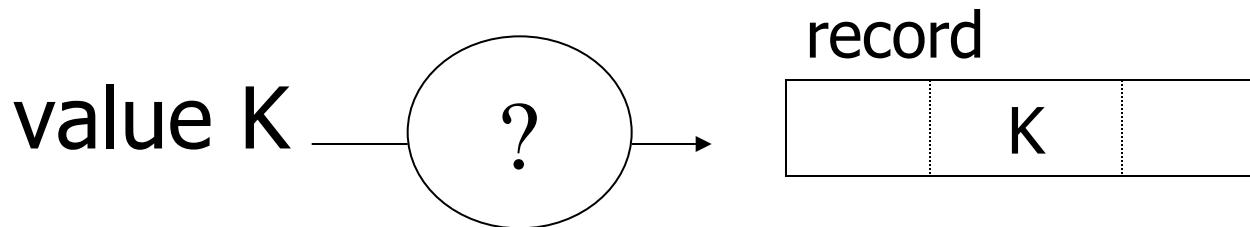
add overflow block:



Indexing (B+ Trees)

based on lecture notes by Hector
Garcia-Molina

Indexing



Index and Data File

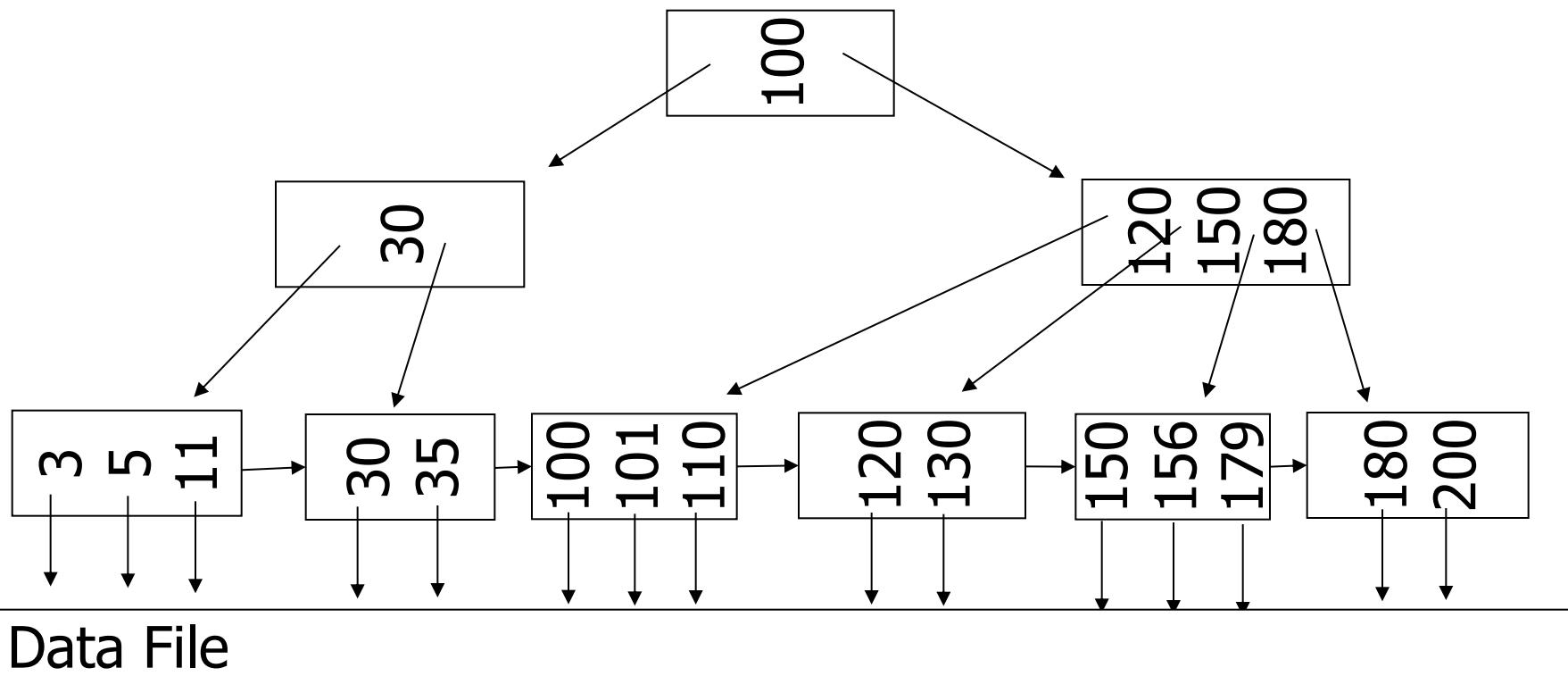
- Index and Data File are separate data structures
- Index is much smaller than Data File. $|Index| << |Data\ File|$
- Index is stored in a collection of blocks in secondary memory
- Data File stored in a collection of blocks in secondary memory
- A key value K in the Index references a single or multiple records in the Data File with that key value K
- Records with the same K value are chained in the Data File (possibly in sequential blocks)
- Operations (search, insert, delete) on indexed data file require blocks from Index and Data File to be moved between primary and secondary memory
- Time complexity measured in terms of number of I/O operations

B+Tree Example

$n=3$

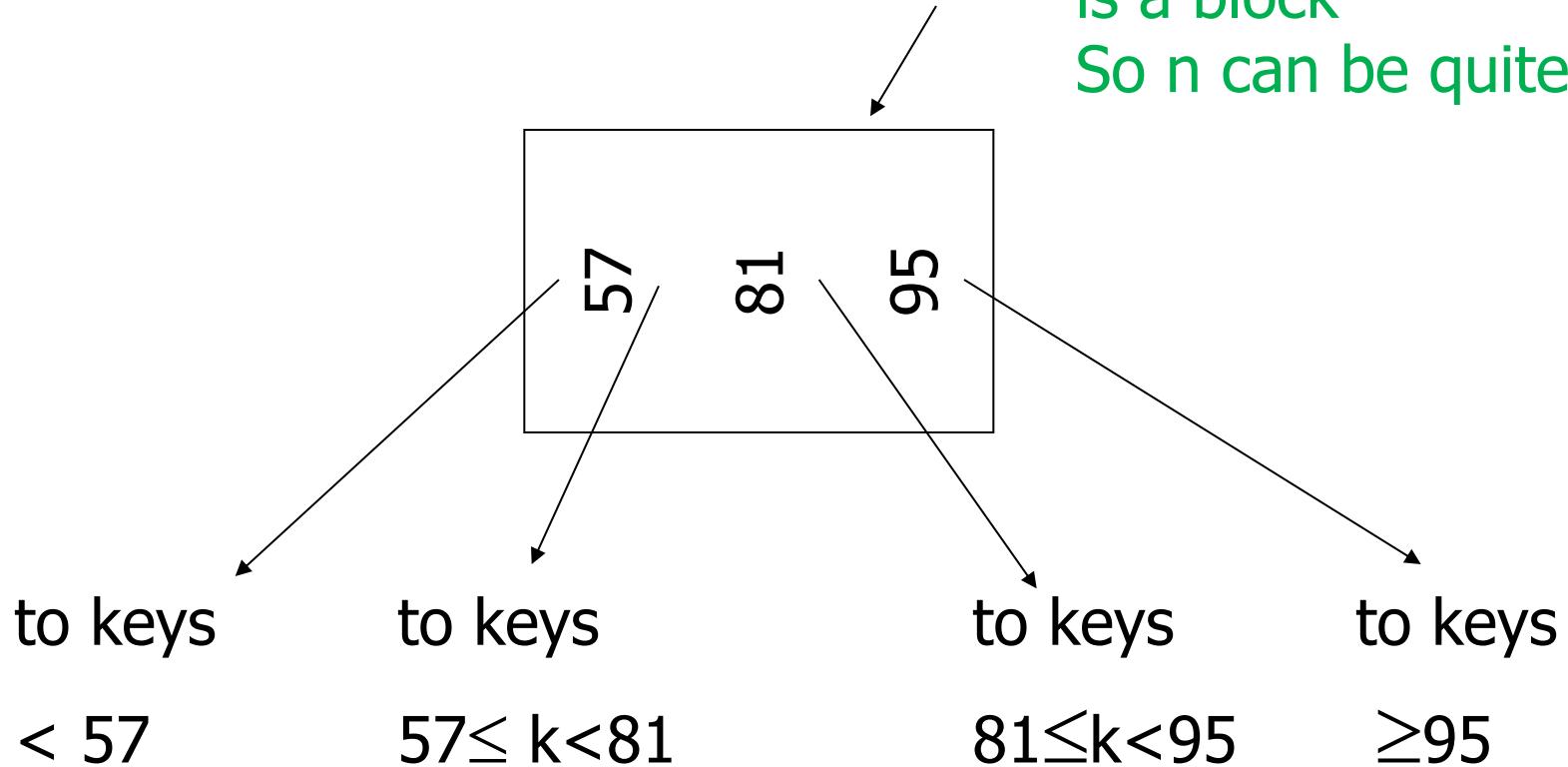
n order of B+ tree

Root

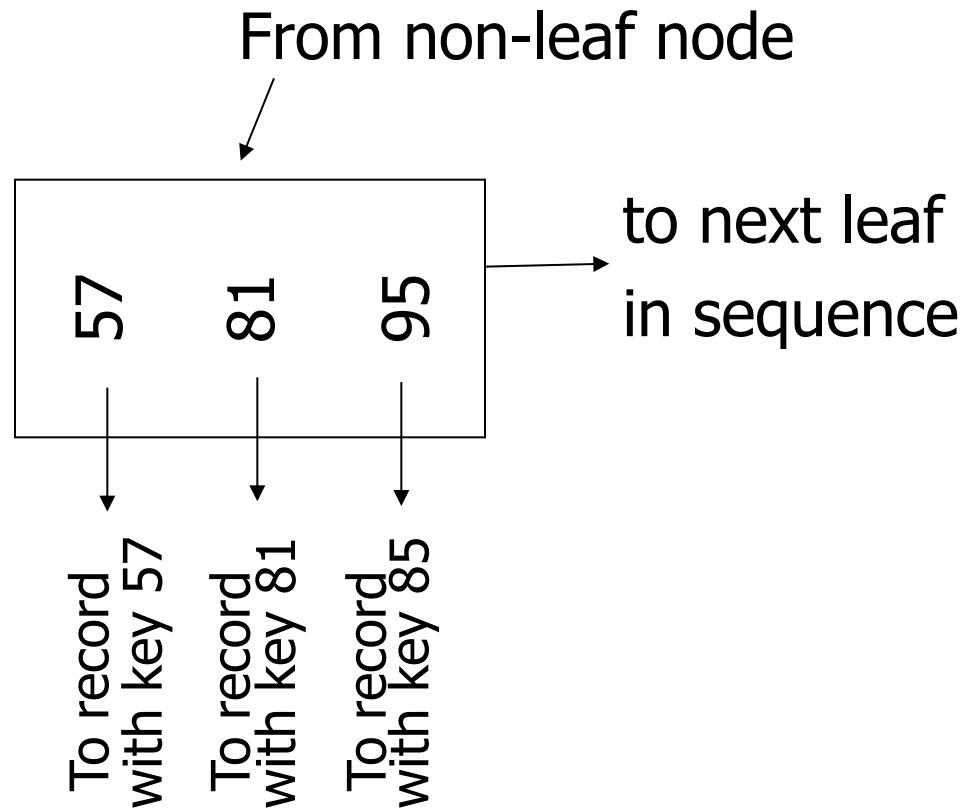


Sample non-leaf

Each node in a B+tree
is a block
So n can be quite large



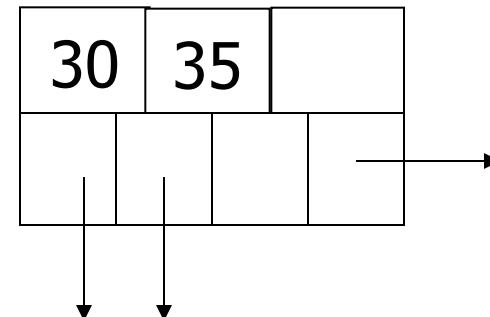
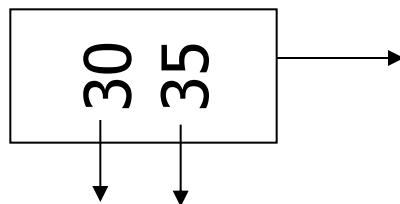
Sample leaf node:



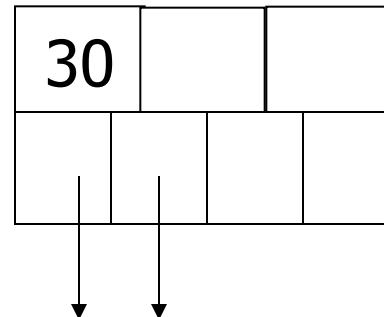
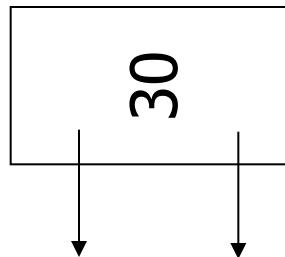
In textbook's notation

$n=3$

Leaf:



Non-leaf:



Size of nodes:

{	n+1 pointers	<u>(fixed)</u>
	n keys	

Observe that a pointer is a block address

$$(n + 1) * |\text{blockaddress}| + n * |\text{key}| \leq \text{blocksize}$$

$$n \leq \frac{\text{blocksize} - |\text{blockaddress}|}{|\text{blockaddress}| + |\text{key}|} \leq \frac{\text{blocksize}}{|\text{blockaddress}| + |\text{key}|}$$

Example: determination of n

- blocksize = 4096 bytes
- $|blockaddress| = 8 \text{ bytes}$
- $|key| = 9 \text{ bytes}$
- $n \leq \frac{\text{blocksize} - |blockaddress|}{|blockaddress| + |key|}$
- Thus n is maximally 240
- $|blockaddress| = 8 \text{ bytes permits } 2^{32} = 4,294,967,296 \text{ blocks to be referenced}$
- $|blockaddress| = 10 \text{ bytes permits 1 trillion blocks to be referenced}$

Don't want nodes to be too empty

- Use at least

Non-leaf: $\lceil (n+1)/2 \rceil$ pointers

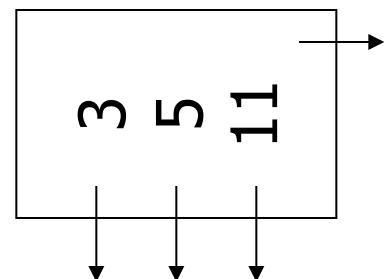
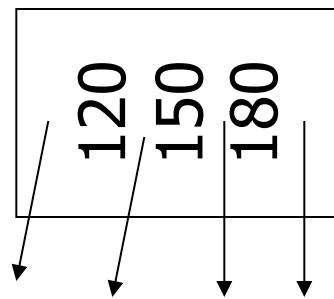
Leaf: $\lfloor (n+1)/2 \rfloor$ pointers to data
+ 1 pointer to next leaf

n=3

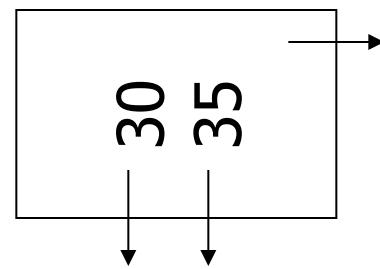
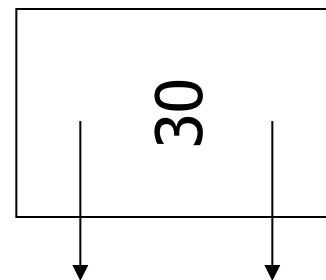
Non-leaf

Leaf

Full node



min. node



B+tree rules tree of order n

- (1) All leaves at same lowest level
(balanced tree)
- (2) Pointers in leaves point to records
except for “next leaf pointer”
- (3) Root must have at least one key
and two pointers

(3) Number of pointers/keys for B+tree

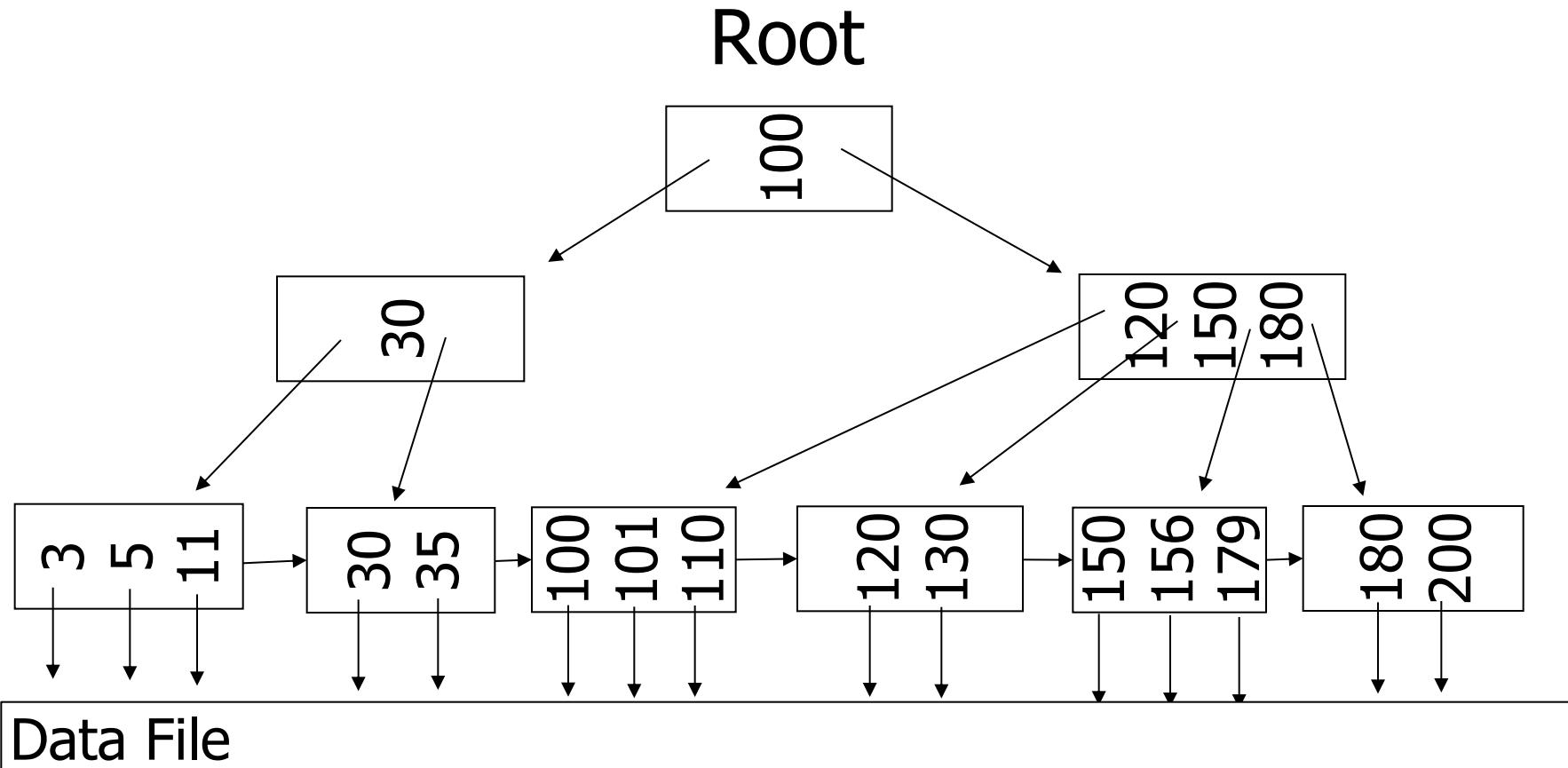
	Max ptrs	Max keys	Min ptrs	Min keys
Non-leaf (non-root)	$n+1$	n	$\lceil(n+1)/2\rceil$	$\lceil(n+1)/2\rceil - 1$
Leaf (non-root)	$n+1$	n	$\lfloor(n+1)/2\rfloor$	$\lfloor(n+1)/2\rfloor$
Root	$n+1$	n	2	1

This slide assume that the tree has at least two levels

B+Tree Search

$n=3$

n order of B+ tree

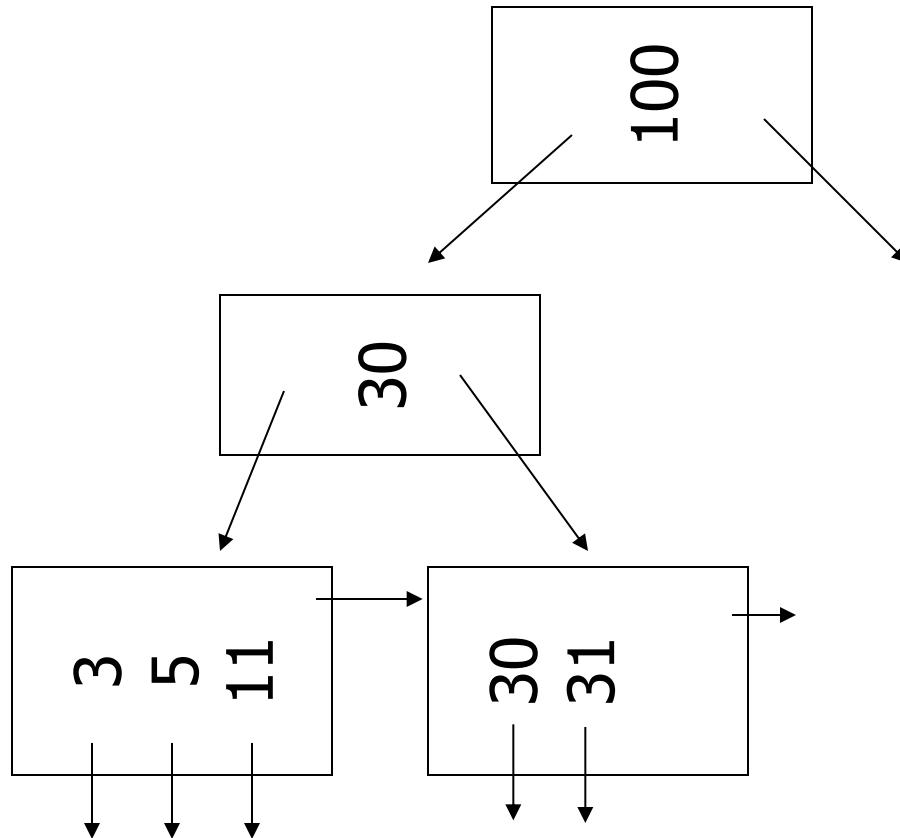


Insert into B+tree

- (a) simple case
 - space available in leaf
- (b) leaf overflow
- (c) non-leaf overflow
- (d) new root

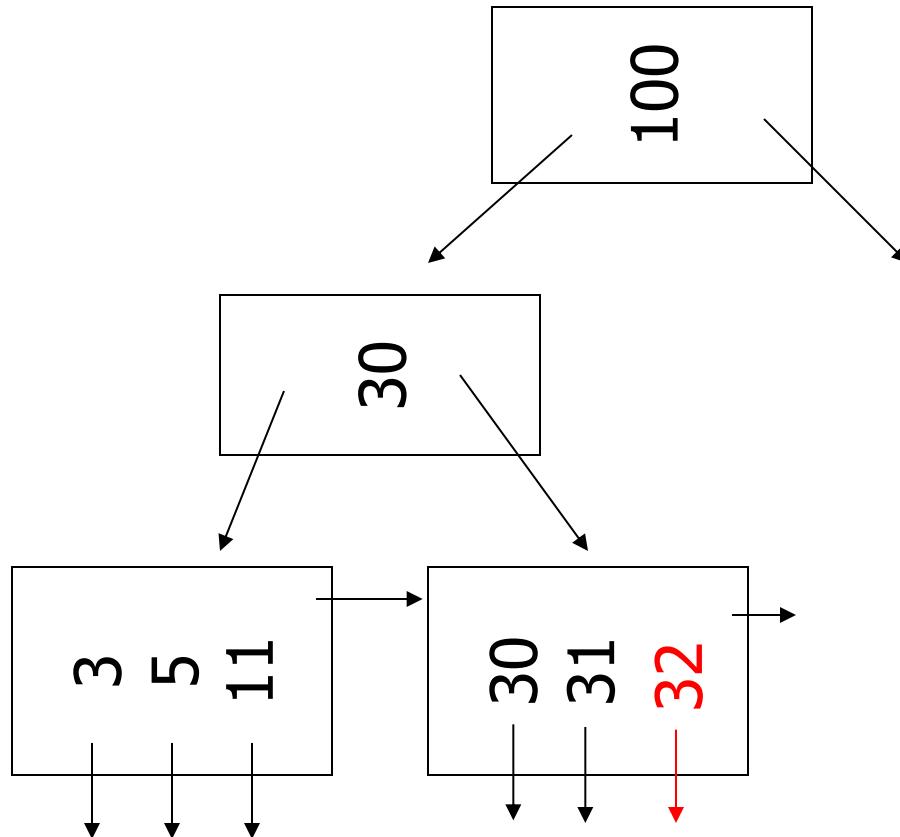
(a) Insert key = 32

n=3



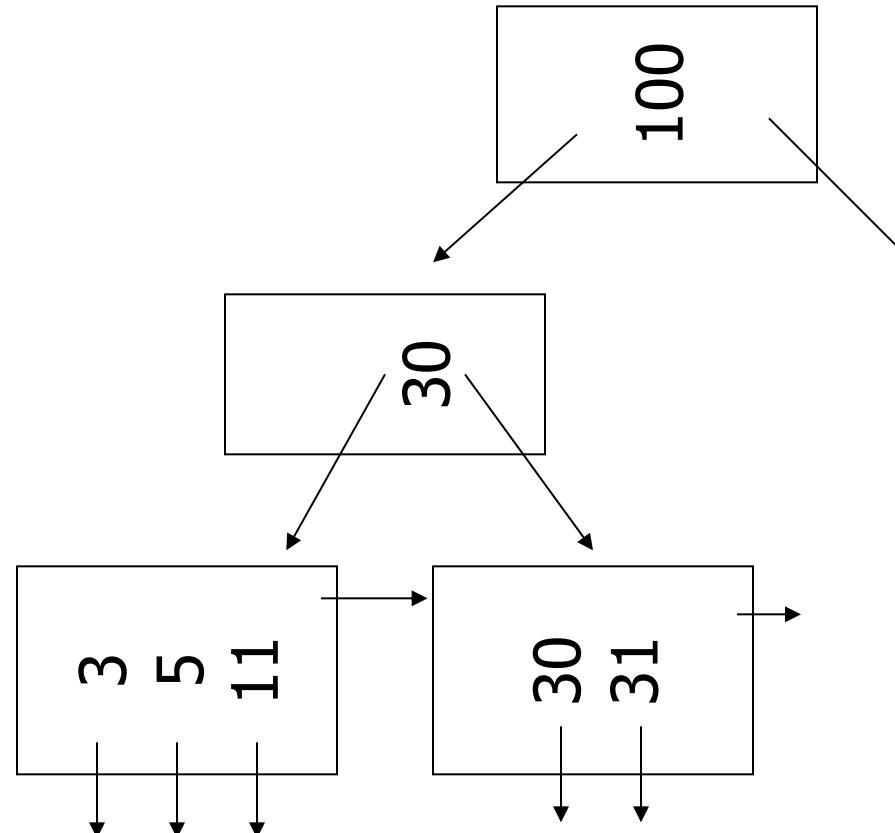
(a) Insert key = 32

n=3



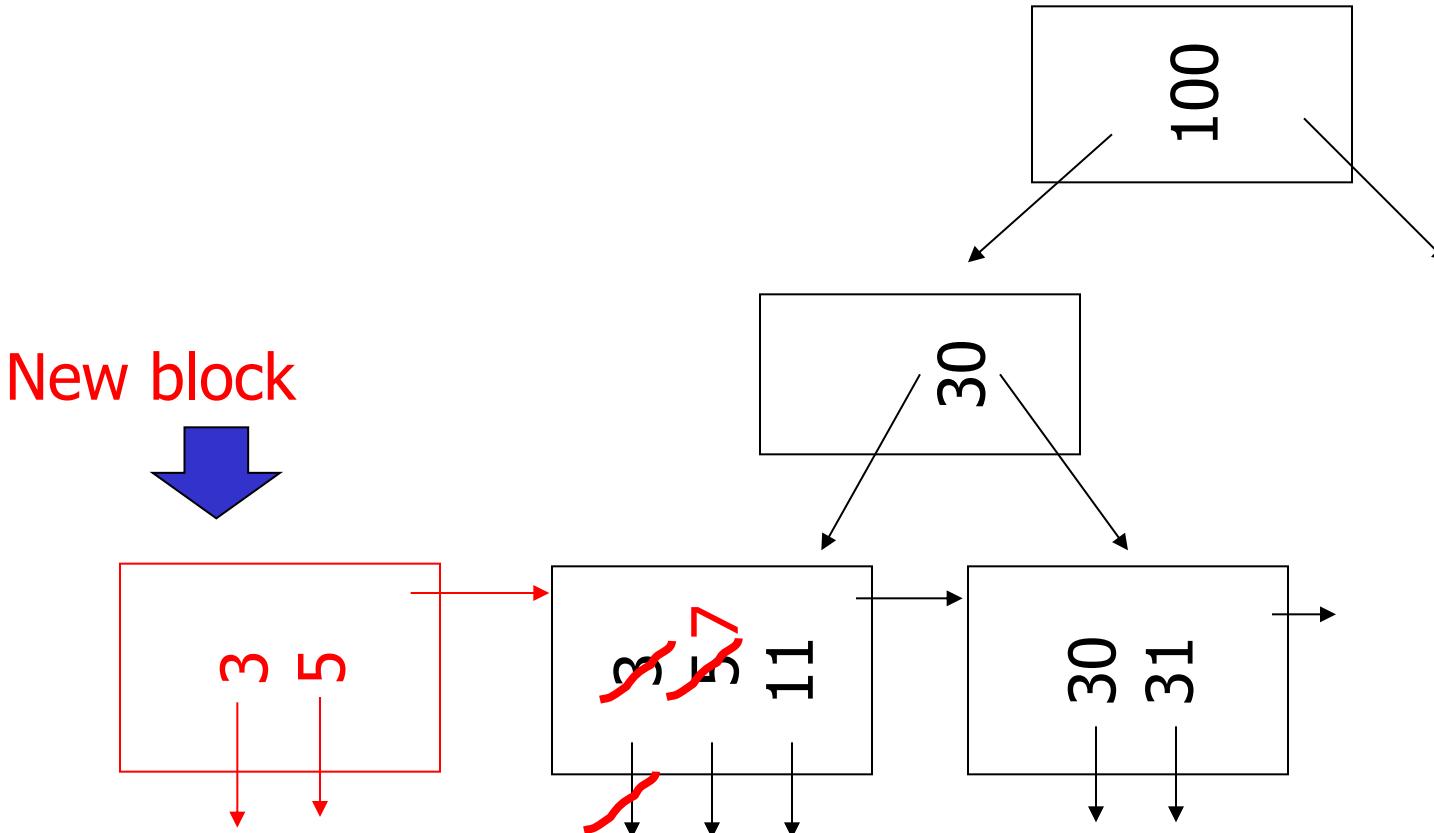
(a) Insert key = 7

n=3



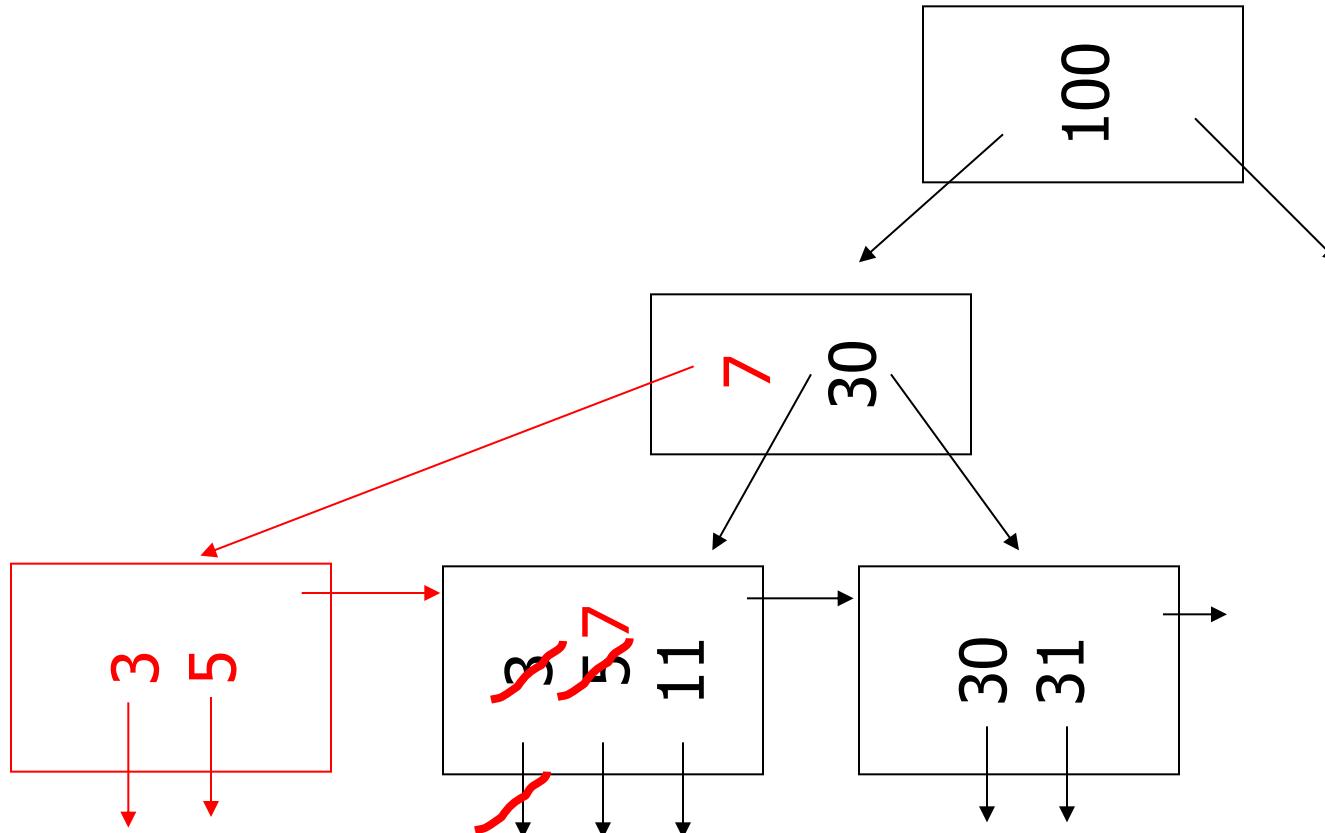
(a) Insert key = 7

n=3



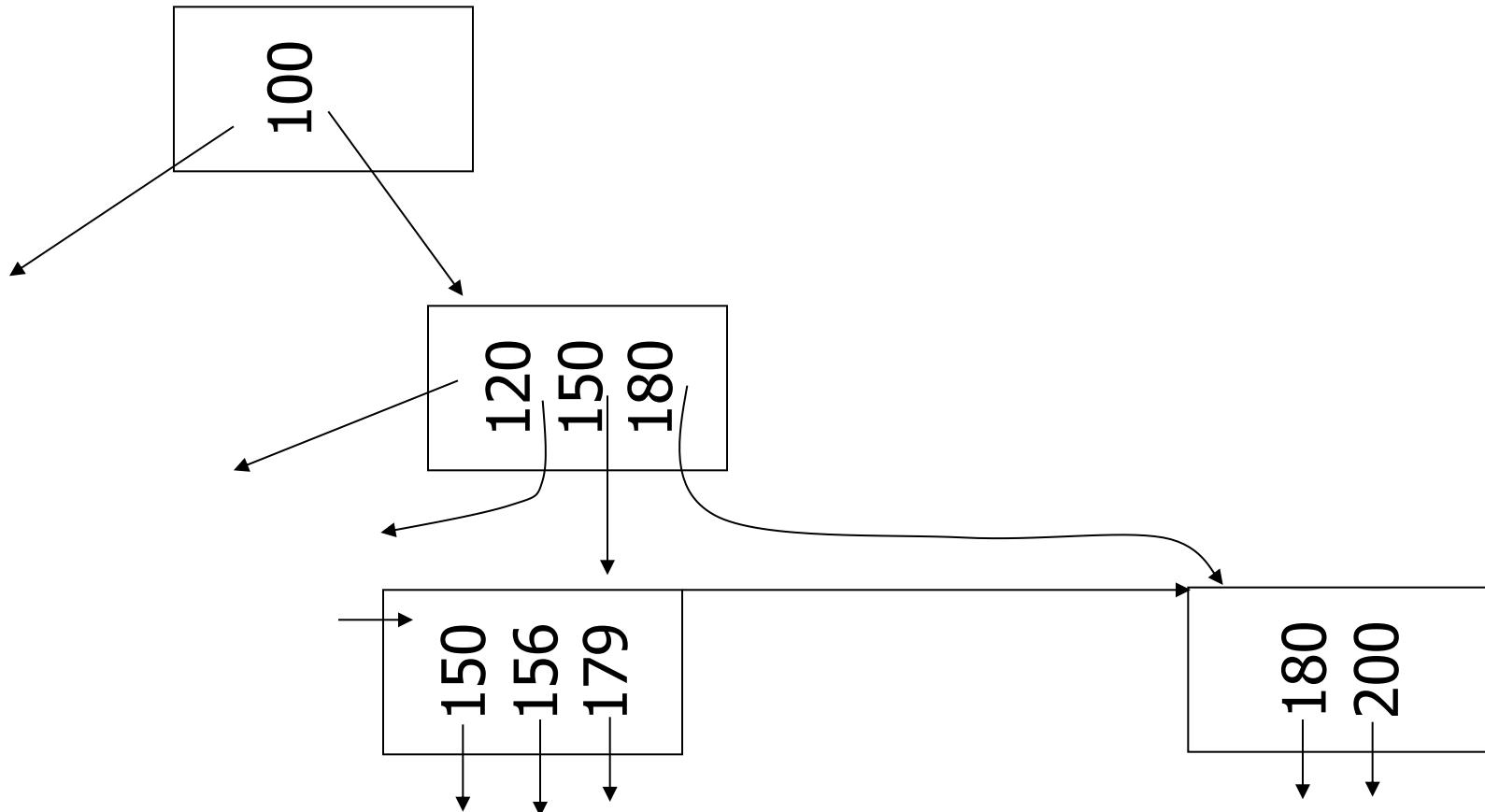
(a) Insert key = 7

n=3



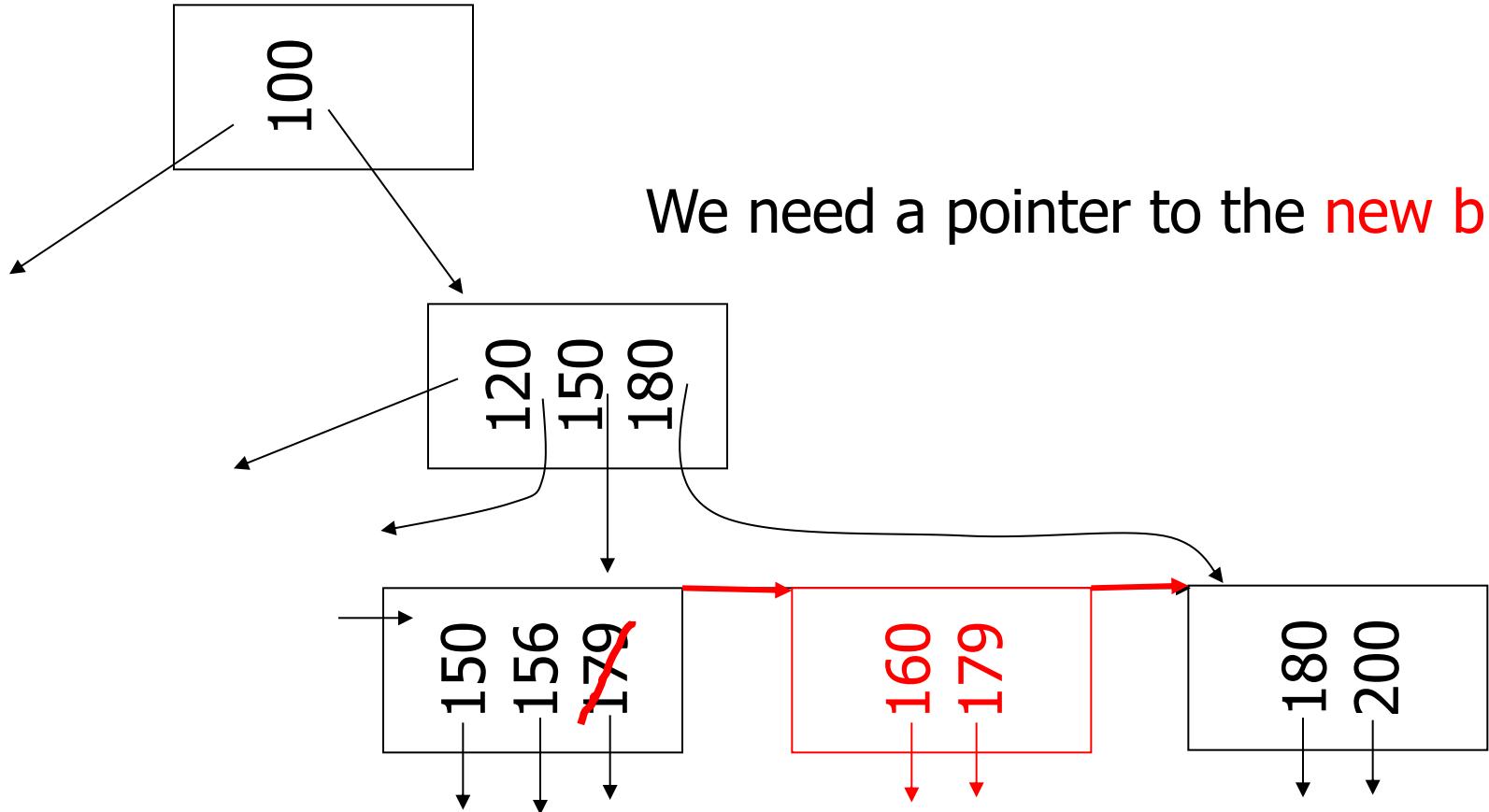
(c) Insert key = 160

n=3



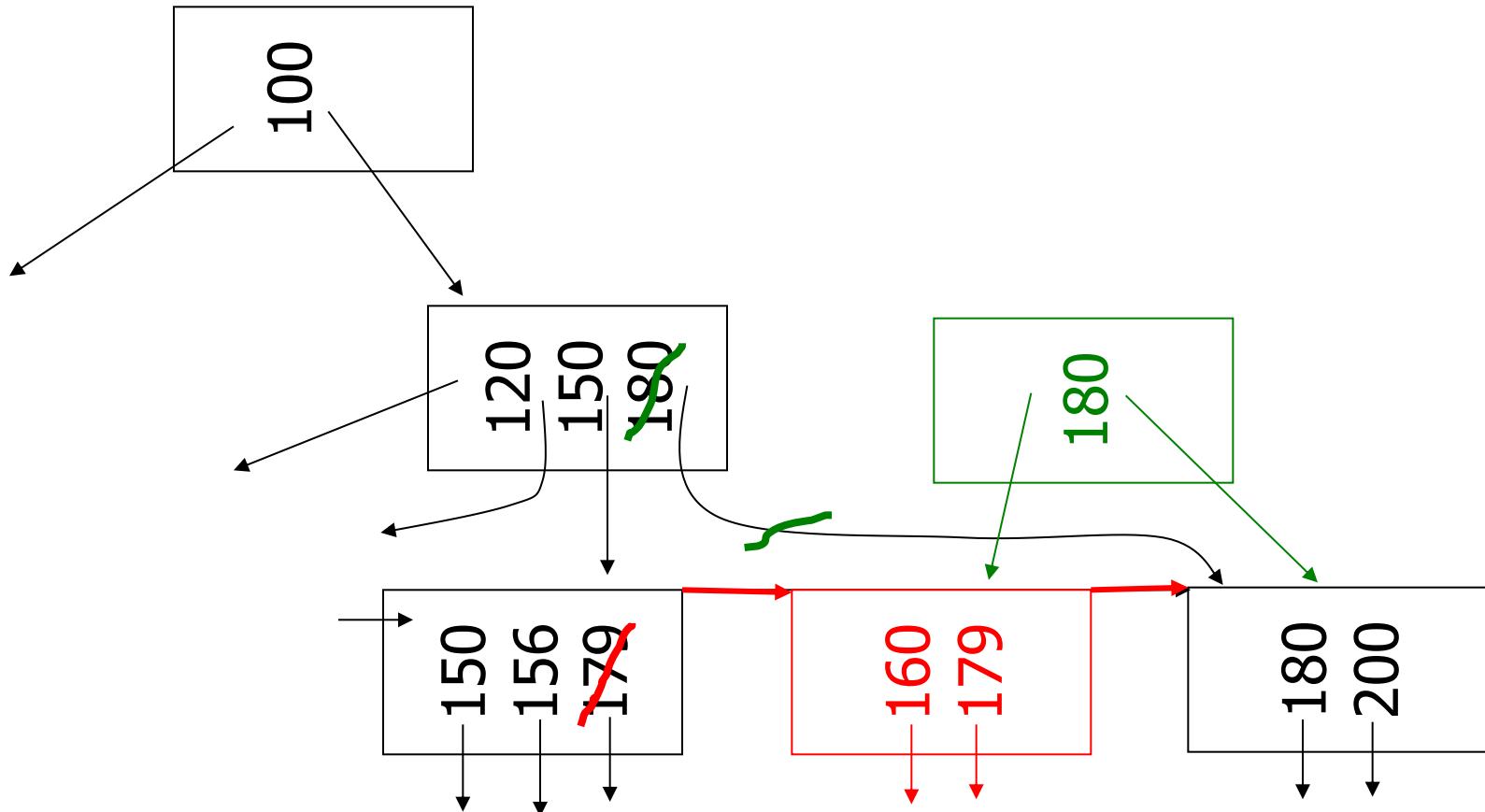
(c) Insert key = 160

n=3



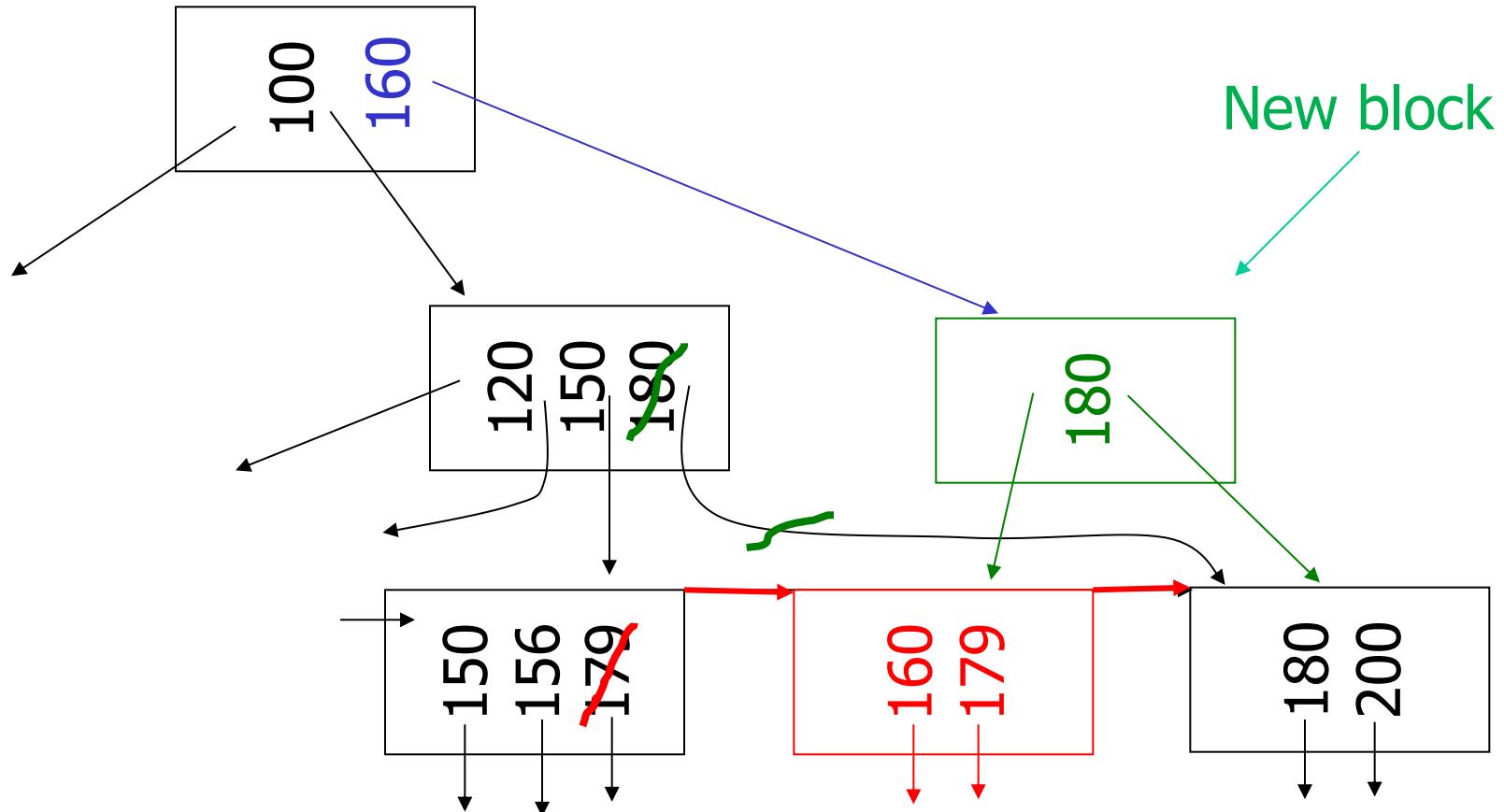
(c) Insert key = 160

n=3



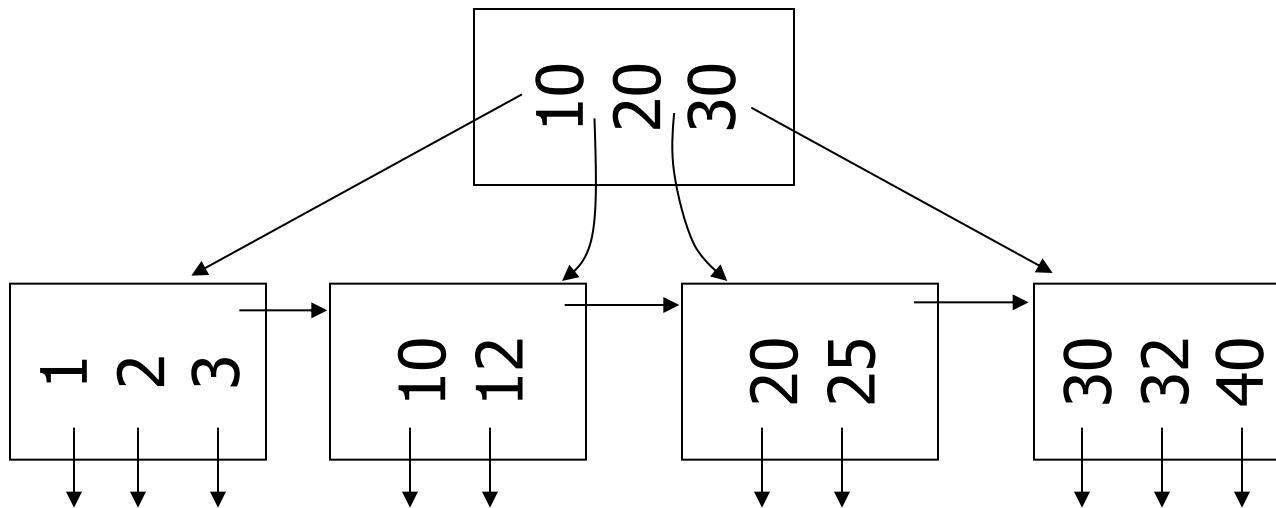
(c) Insert key = 160

n=3



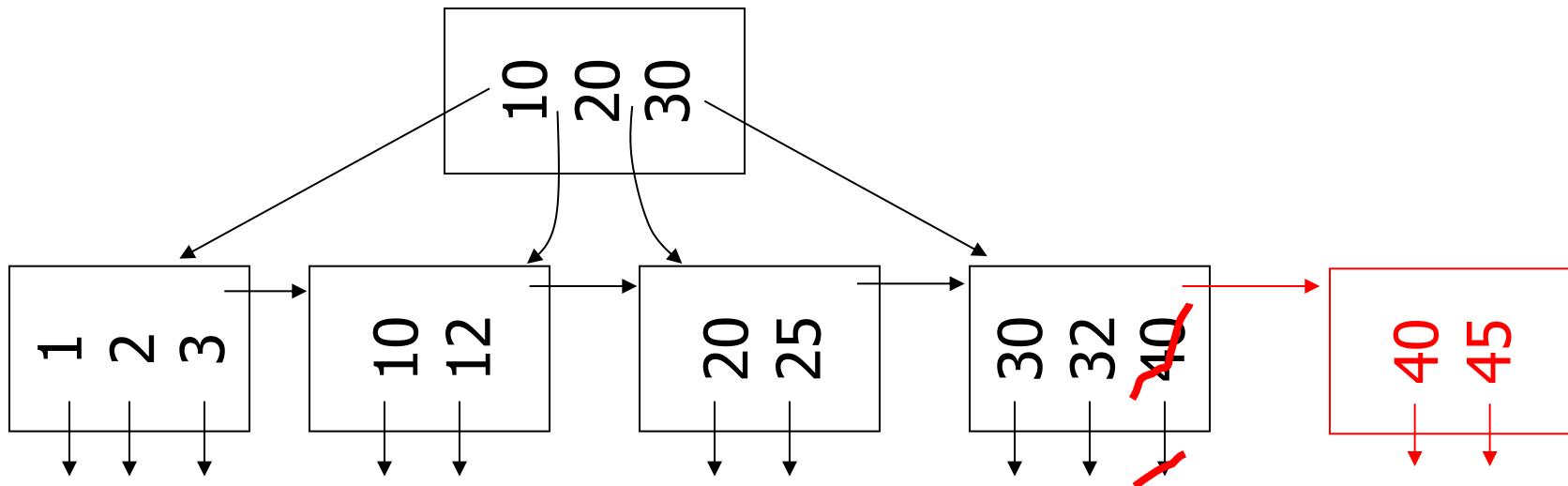
(d) New root, insert 45

n=3



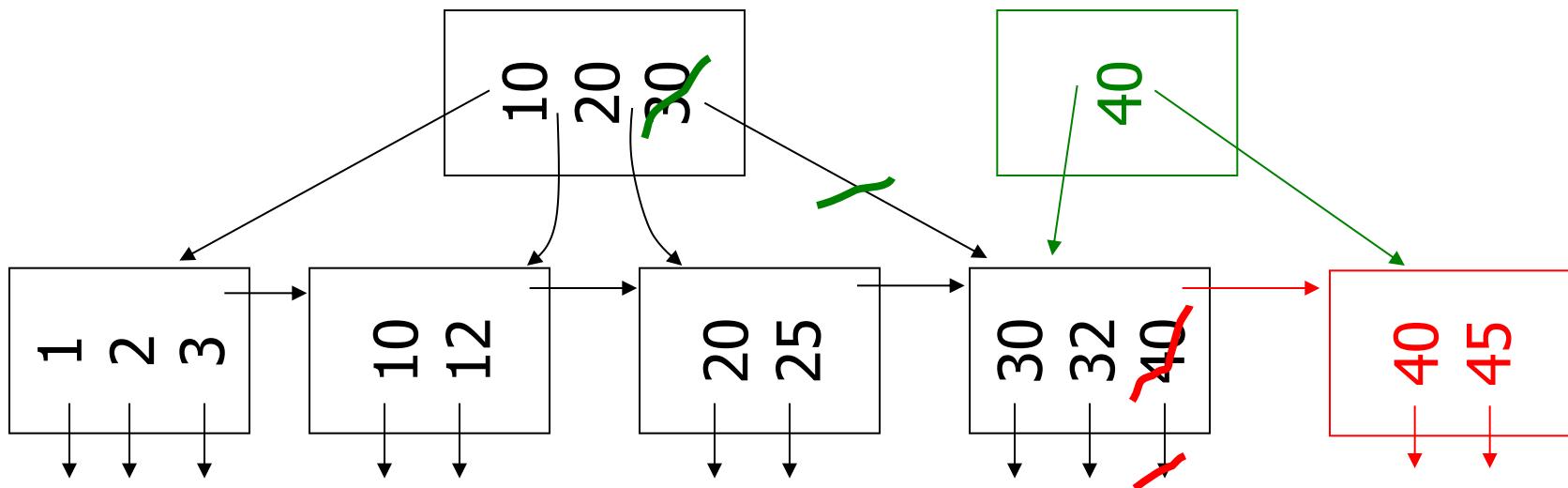
(d) New root, insert 45

n=3



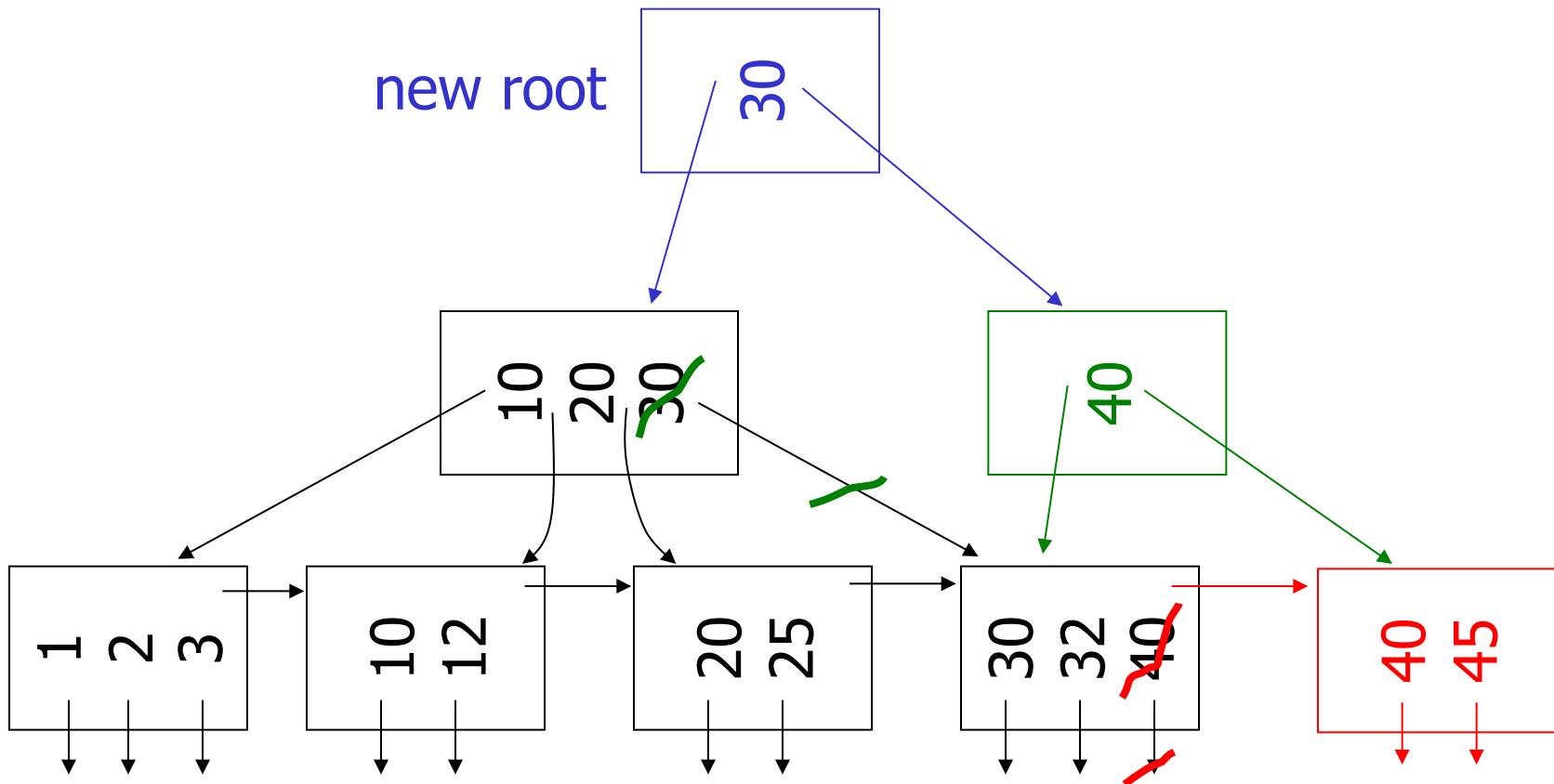
(d) New root, insert 45

n=3



(d) New root, insert 45

n=3



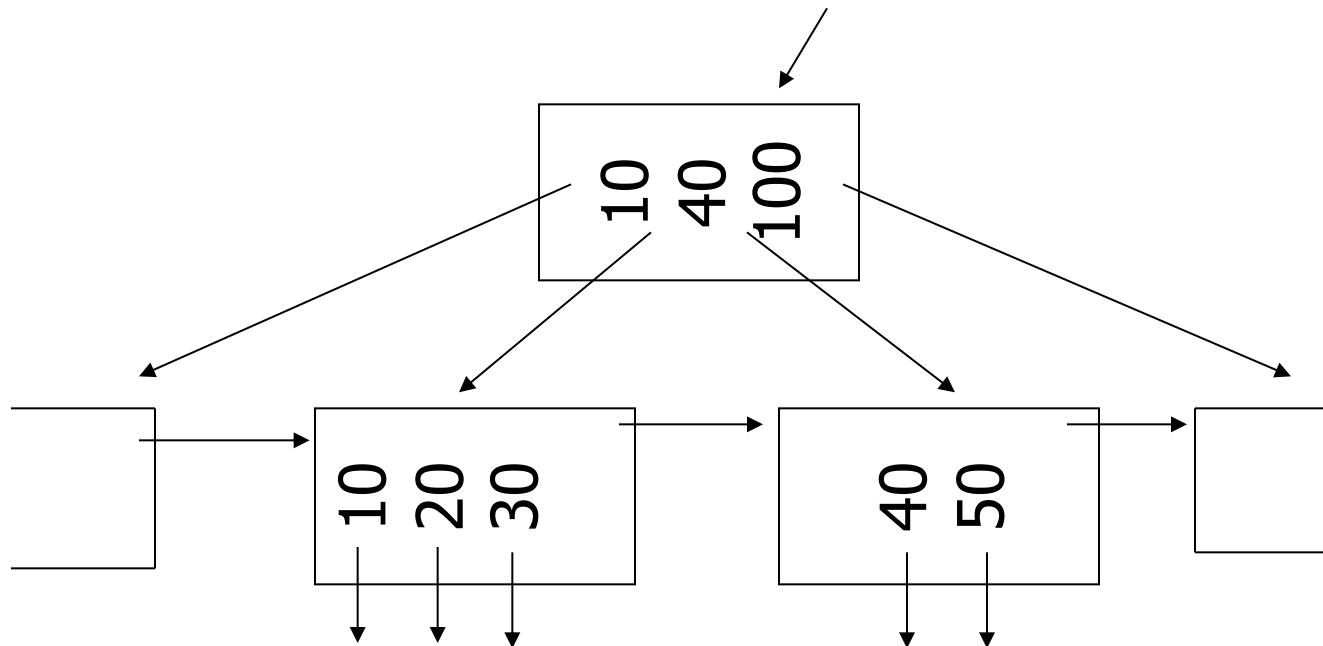
Deletion from B+tree

- (a) Simple case - no example
- (b) Coalesce with neighbor (sibling)
- (c) Re-distribute keys
- (d) Cases (b) or (c) at non-leaf

(b) Coalesce with sibling

– Delete 50

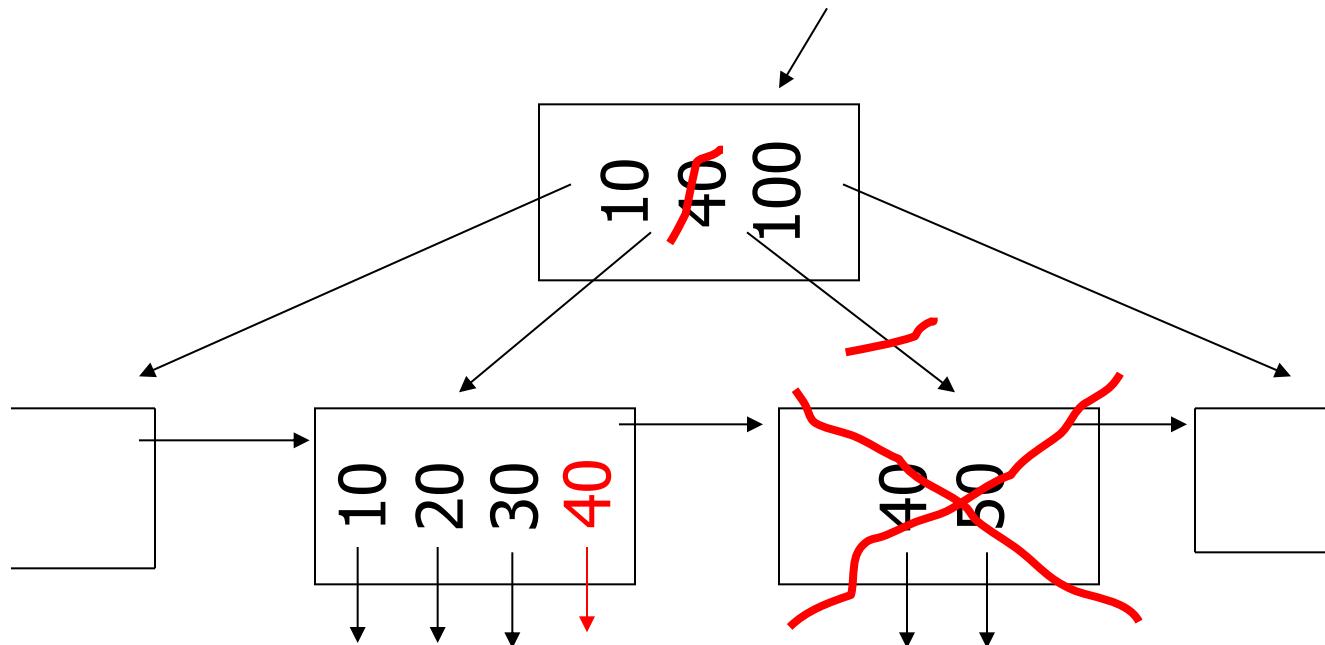
n=4



(b) Coalesce with sibling

– Delete 50

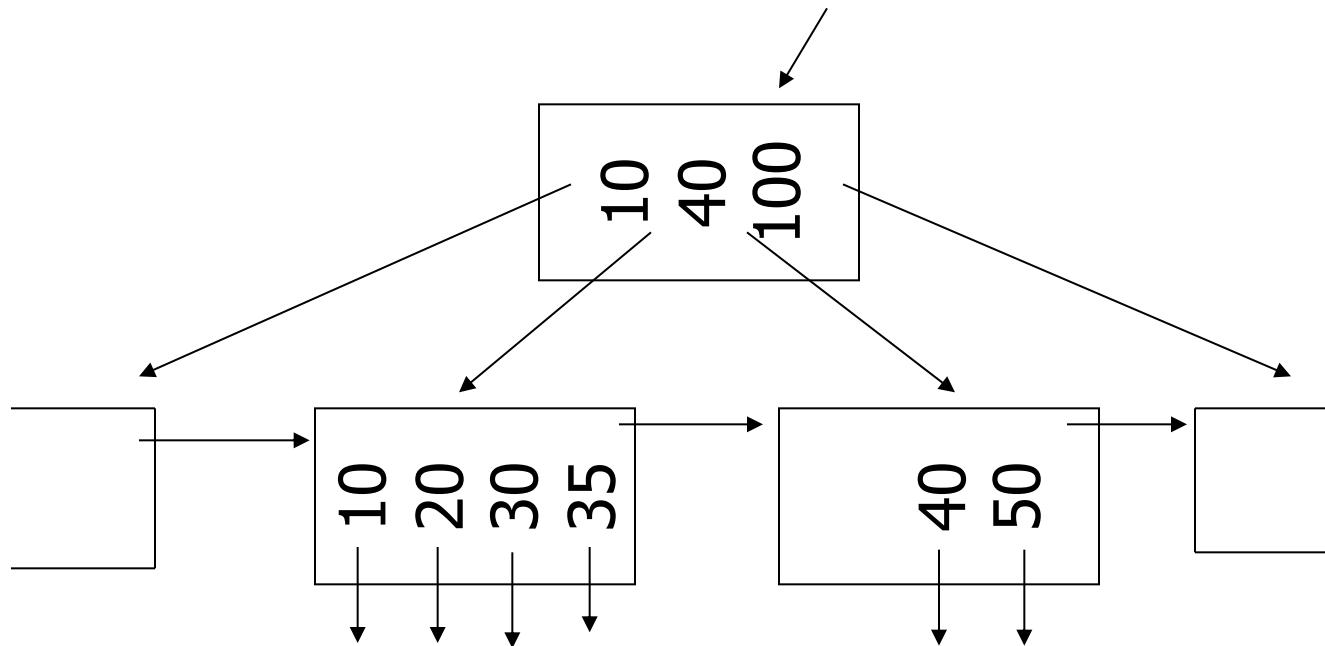
n=4



(c) Redistribute keys

- Delete 50

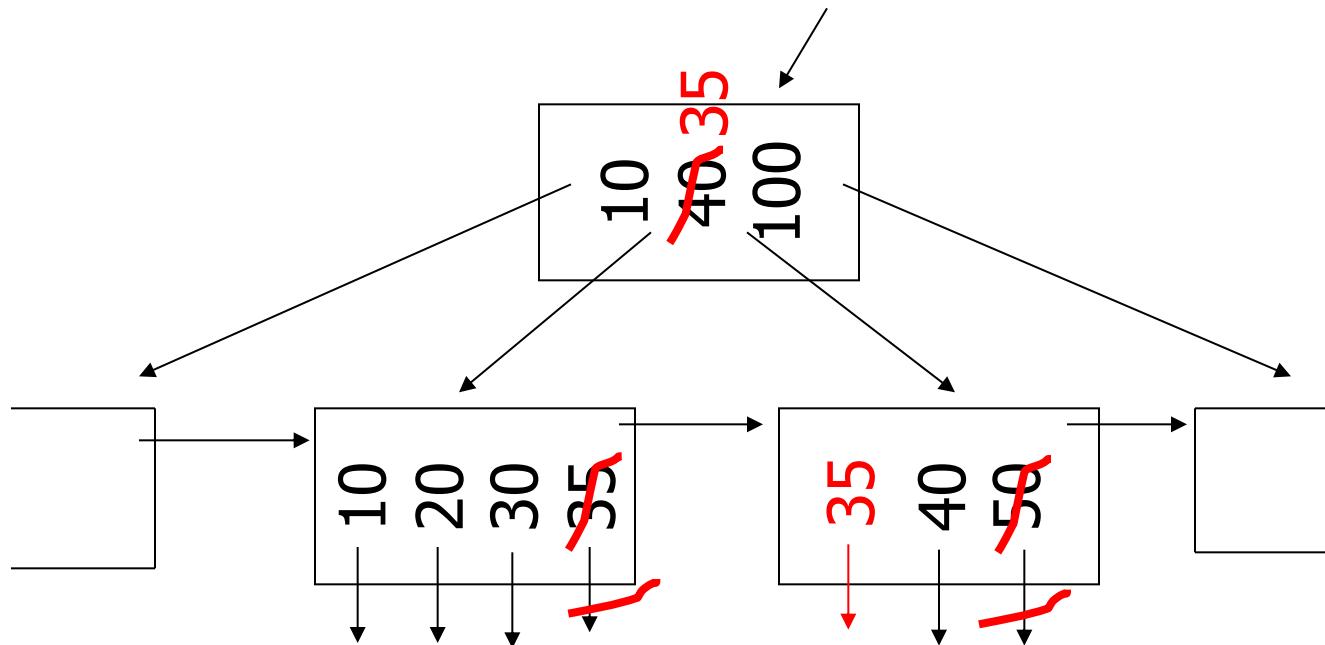
n=4



(c) Redistribute keys

- Delete 50

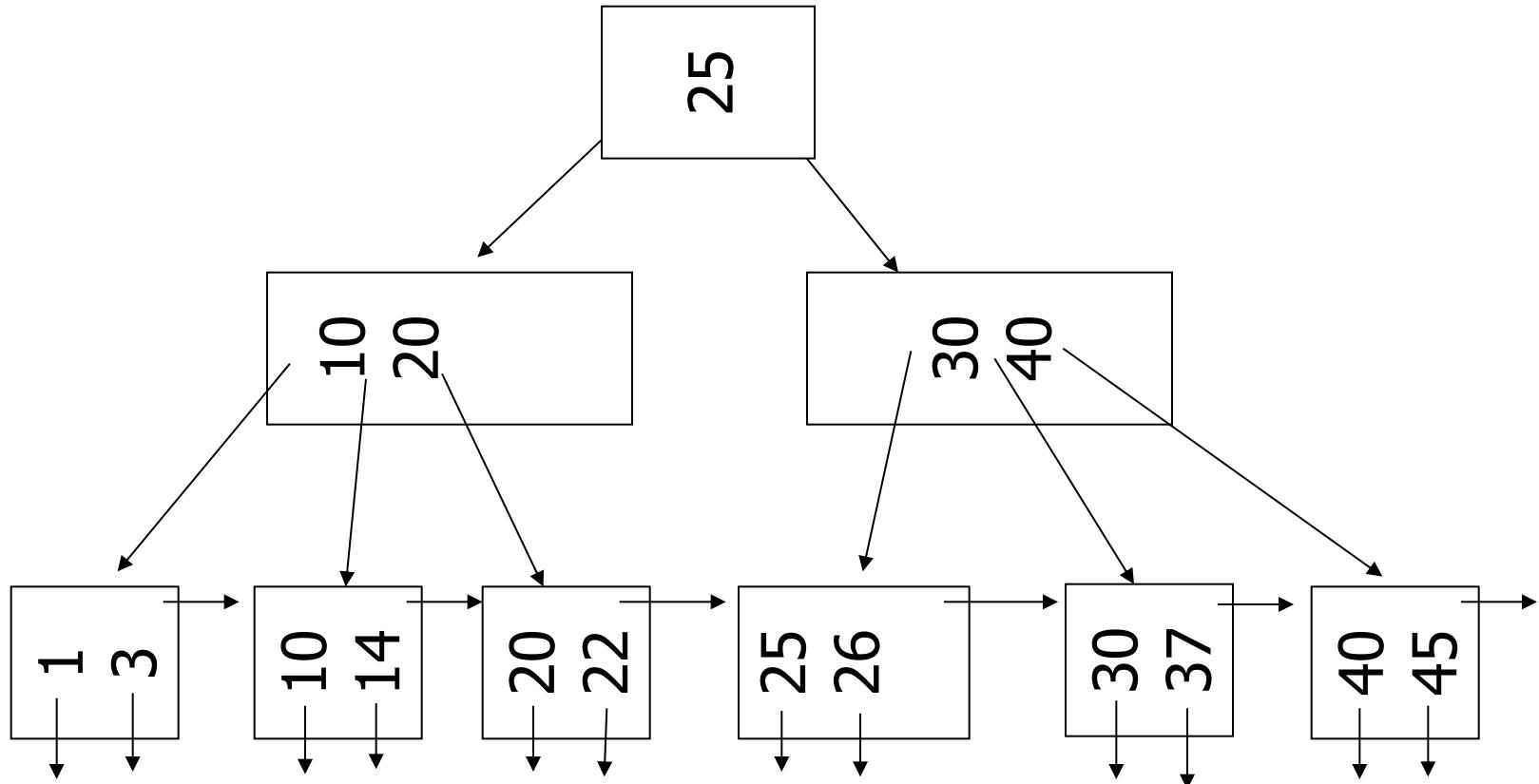
n=4



(d) Non-leaf coalesce

– Delete 37

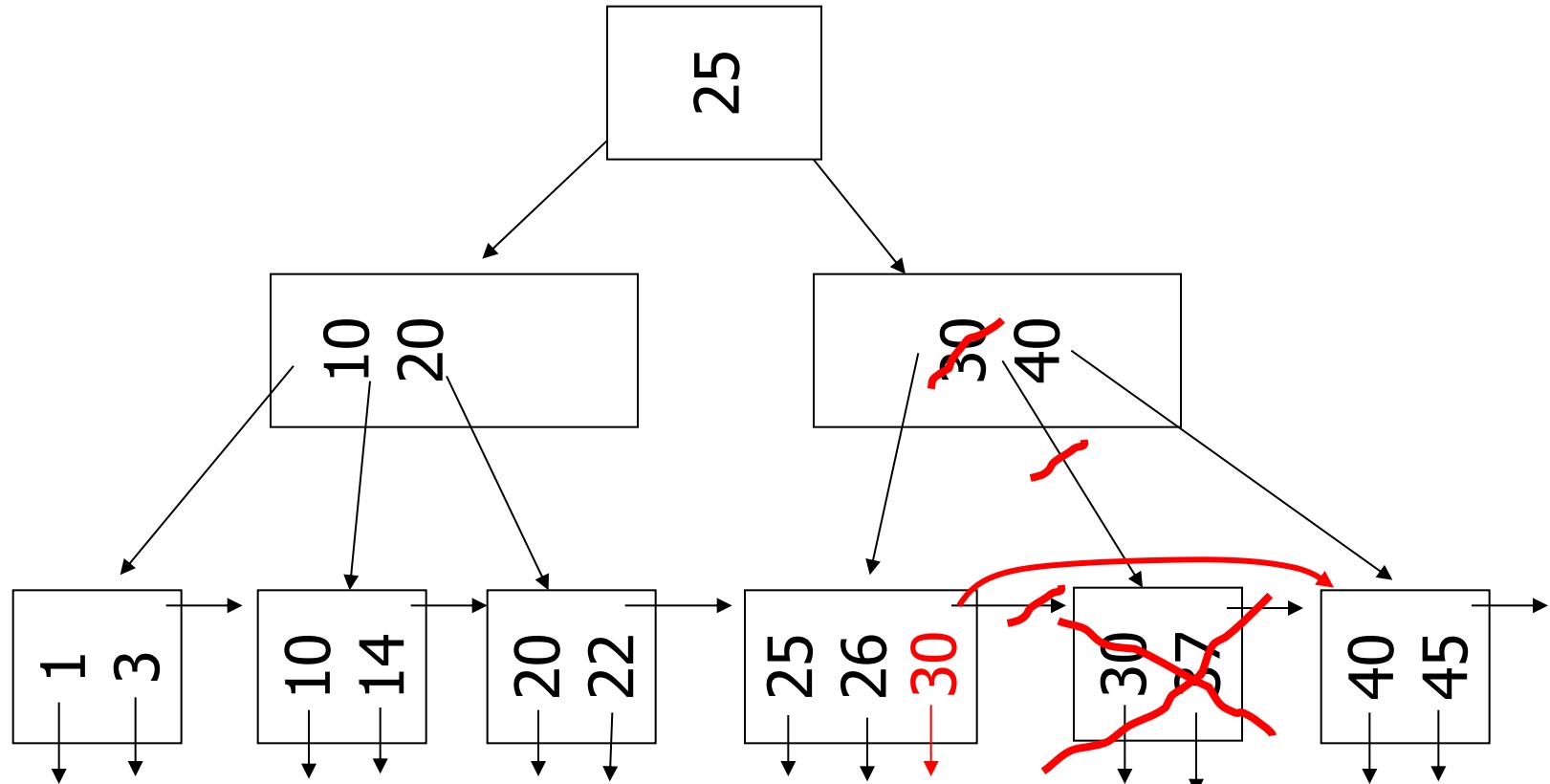
n=4



(d) Non-leaf coalesce

– Delete 37

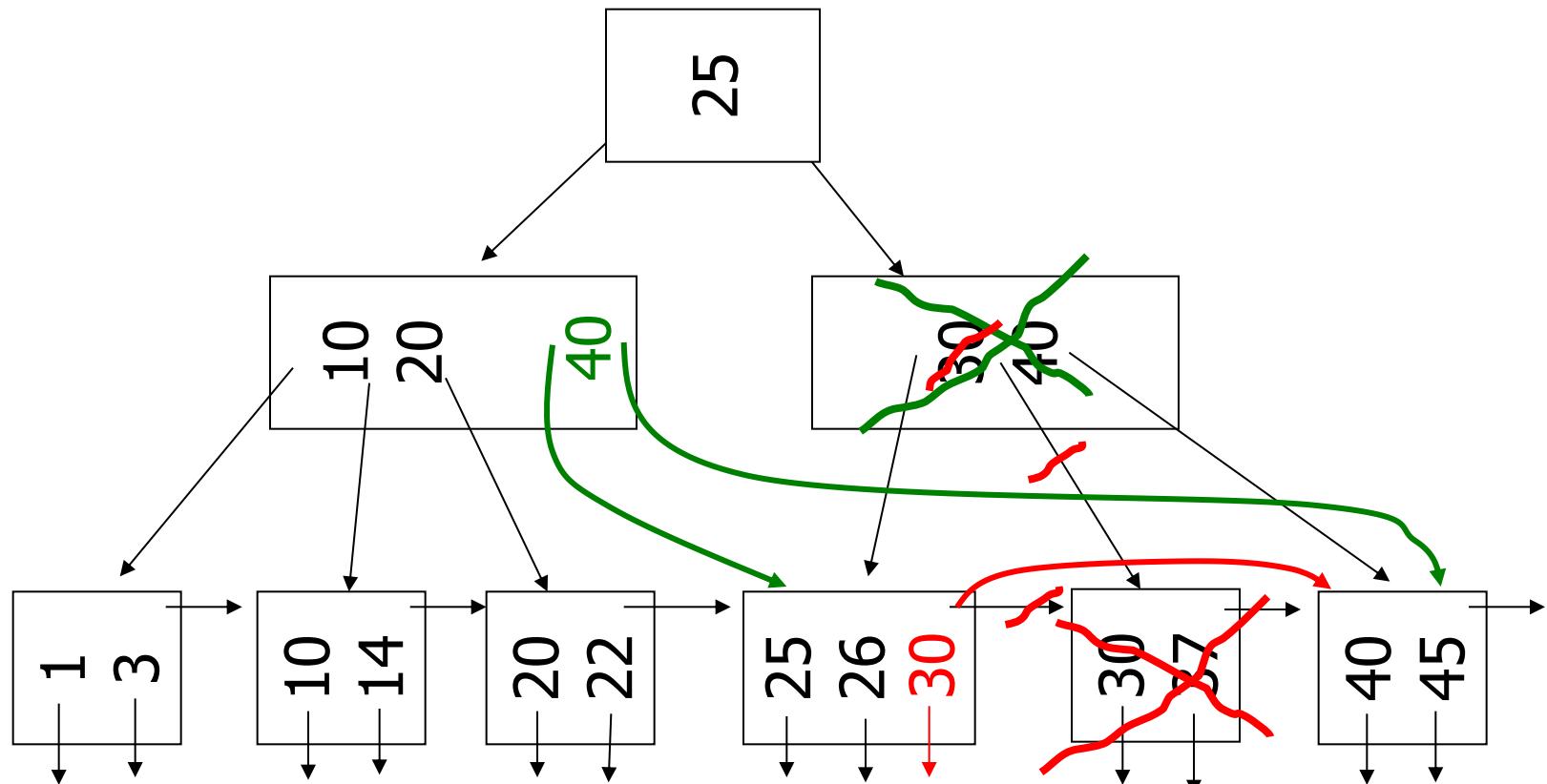
n=4



(d) Non-leaf coalesce

– Delete 37

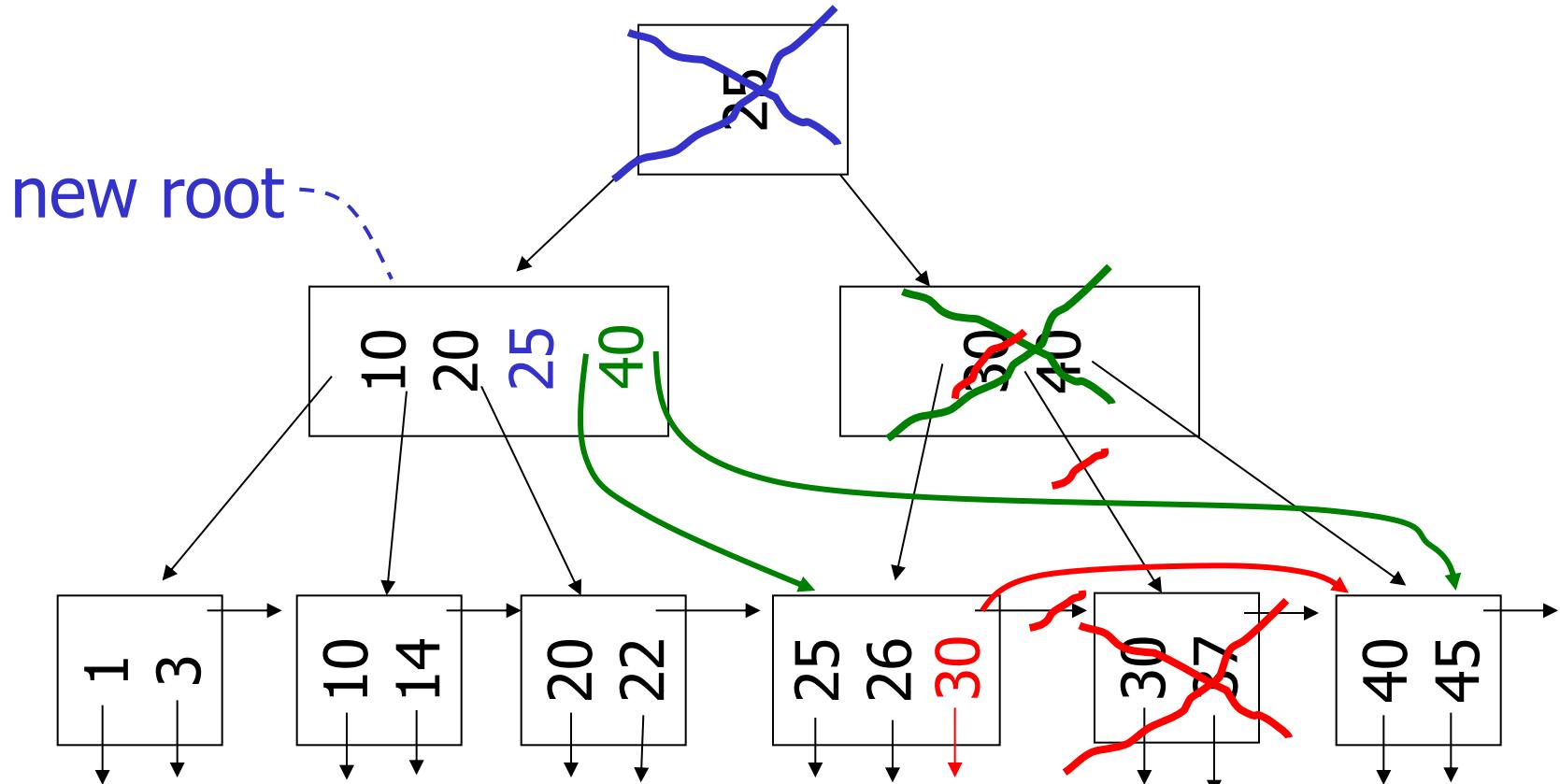
n=4



(d) Non-leaf coalesce

– Delete 37

n=4



Complexity analysis of search

Assumption: N blocks in the index
n order or the tree

Each search requires navigating along a path
from the root to a leaf

Thus the complexity corresponds to the
height **h** of the tree

height **h** is maximum if the branching factor at the nodes is minimal

at the root: 2

at non-leaf nodes: essentially $n/2$

the root splits the tree into two trees of $N/2$ keys

we must find the number **a** such that $(\frac{n}{2})^a \geq N/2$

So $a \sim \log_{n/2}(N/2) = O(\log_n(N))$

If the branching factor is maximum at each nodes,
the complexity analysis also gives $O(\log_n(N))$

Complexity analysis for insert and delete

In the worst case, for both insert and delete, processing is determined by a downward phase of h steps and an upward phase also of h steps

Consequently, the complexity of insert and delete is also $O(\log_n(N))$

To improve complexity, place the first several levels in main memory

For typical cases, search time is measured in terms of a few block I/O's and this for very large data files

The leaf level provides a sorted list of the records in the data file.

Range searches can be accommodated: given range (k_1, k_2) , locate the leaf holding k_1 and then follow along the leaf level until reaching records with key value higher than k_2

Algorithms for Relational Algebra Operations (Join Algorithms)

Overview of Join Algorithms

- Nested-loop and block nested-loop join $R \bowtie_{\theta} S$
- Sort-merge join $R \bowtie S$
- Hash-based join strategies $R \bowtie S$
- Index join $R \bowtie S$

Join Operator

- JOIN: Most important relational operator
 - Potentially very expensive
 - Required in all practical queries and applications
 - Often appears in groups of joins
- Example: Relations R (A, B) and S (B, C)

```
SELECT *
FROM R JOIN S ON (R.B  $\theta$  S.B)
with  $\theta$ : =,  $\neq$ ,  $<$ ,  $\leq$ ,  $>$ ,  $\geq$ 
```

Unary versus Binary Operations

- Relational operators working on one table
 - Selection, projection
- On two tables
 - Product, Join, Semi-join, Intersection, Union, Difference
- **Binary operators** are usually more expensive
 - Unary: Look at table (scanning, index, hash, ...)
 - Binary: Look at each tuple of first table **for each tuple of second table**
 - “Potentially” quadratic complexity

Nested-loop Join

Nested-loop join

```
FOR EACH r IN R DO
    FOR EACH s IN S DO
        IF (r.Bθs.B) THEN OUTPUT (r ⋈θ s)
```

Some improvement (block-based)

```
FOR EACH block x IN R DO
    FOR EACH block y IN S DO
        FOR EACH r in x DO
            FOR EACH s in y DO
                IF (r.Bθs.B) THEN OUTPUT (r ⋈θ s)
```

- Cost estimations
 - $b(R)$, $b(S)$ number of blocks in R and in S , respectively
 - Each block of outer relation is read once
 - Inner relation is **read once for each block** of outer relation
 - Inner **two loops are free** (only main memory operations)
 - Altogether: $b(R)+b(R)*b(S)$

Example

- Assume $b(R)=10,000$, $b(S)=2,000$
- R as outer relation
 - $IO = 10,000 + 10,000*2,000 = 20,010,000$
- S as outer relation
 - $IO = 2,000 + 2,000*10,000 = 20,002,000$
- Use **smaller relation as outer relation**
 - For large relations, choice doesn't really matter
- Can we do better?

Block nested-loop join

- M – Size of main memory in blocks (or number of buffers)
- Rule of thumb: **Use all memory you can get**
 - Use all memory the buffer manager allocates to the process

- **Blocked-nested-loop**

```
FOR i=1 TO b(R) / M DO
    READ NEXT R-chunk of M blocks of R into Memory buffer
    FOR EACH block y IN S DO
        FOR EACH r in R-chunk DO
            FOR EACH s in y do
                IF ( r.B θ s.B) THEN OUTPUT (r ⋈θ s)
```

- Cost estimation
 - Outer relation is read once
 - Inner relation is read once for **every chunk** of R
 - There are $\sim b(R)/M$ chunks
 - IO = $b(R) + b(R)*b(S)/M$

- Example
 - Assume $b(R)=10,000$, $b(S)=2,000$, $M = 500$
 - R as outer relation
 - $IO = 10,000 + 10,000*2,000/500 = 50,000$
 - S as outer relation
 - $IO = 2,000 + 2,000*10,000/500 = 42,000$
 - Compare to one-block NL: 20,002,000 IO
- Use smaller relation as outer relation
- But sizes of relations do matter:
 - If one relation fits into memory ($\min(b(R), b(S)) < M$)
 - Total cost: $b(R) + b(S)$

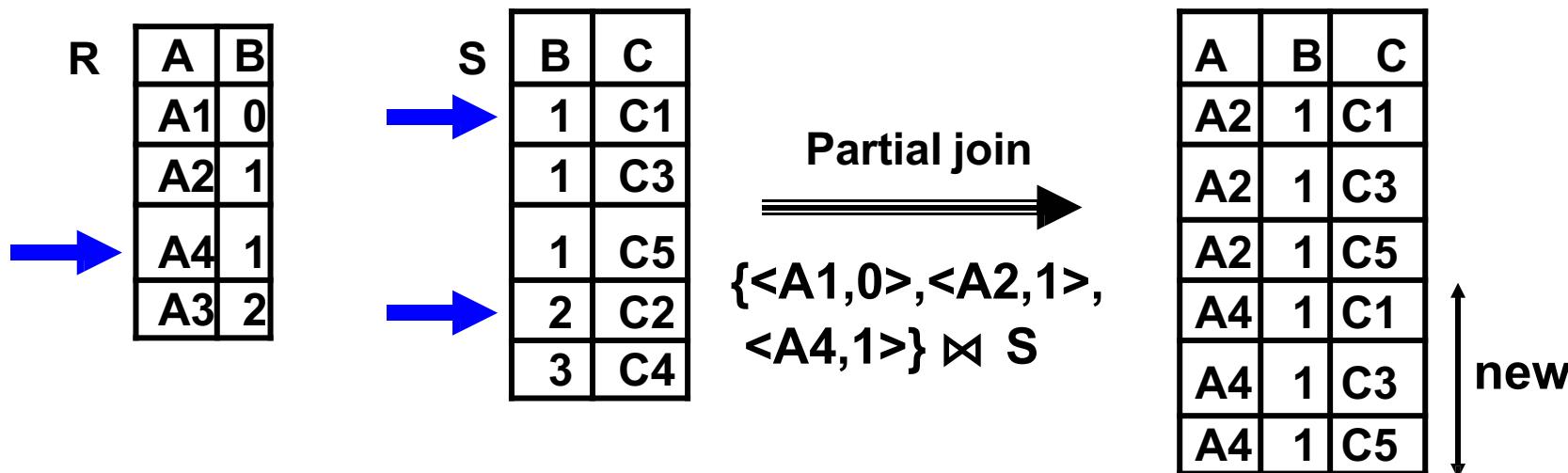
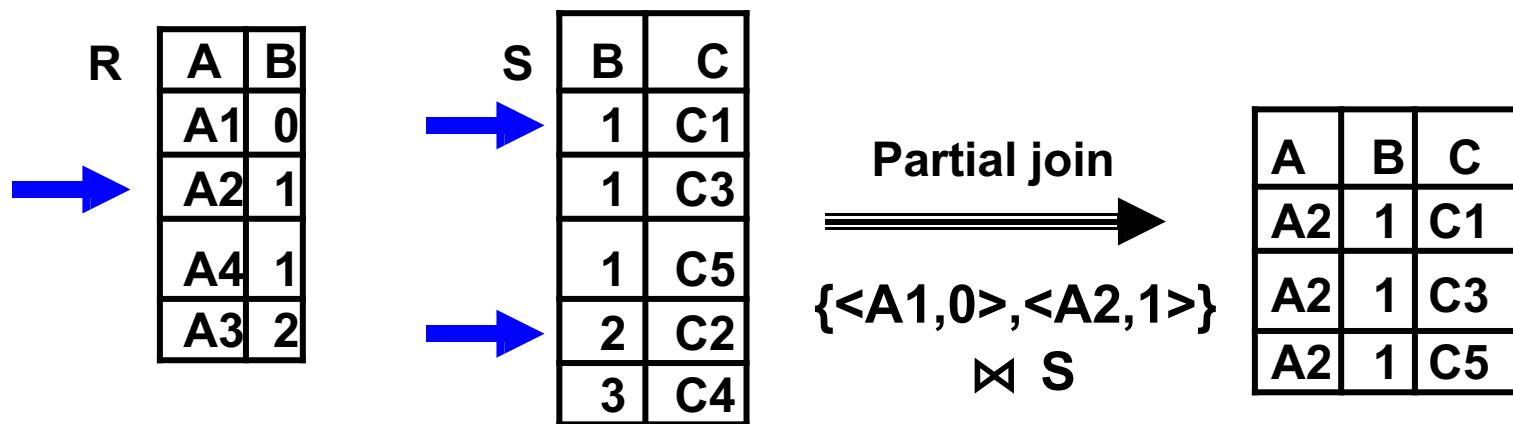
Sort-Merge Join

- How does it work?
- Only works for Natural Join $R \bowtie S$
- What does it cost?
- Does it matter which is outer/inner relation?
- When is it better then block-nested loop?
- Be concerned about skew on join attribute

Sort-Merge Join

- How does it work?
 - Sort both relations on join attribute(s)
 - Merge both sorted relations
- Caution if duplicates (skew) exist
 - The result size still is $|R| * |S|$ in worst case
 - If there are r/s tuples with value x in the join attribute in R / S, we need to output $r * s$ tuples for x
 - More importantly, all these r/s must simultaneously fit in main memory (not always true if there is skew)

Example



Cost estimation (without skew)

- Sorting R costs $2*b(R) * \text{ceil}(\log_M(b(R)))$
- Sorting S costs $2*b(S) * \text{ceil}(\log_M(b(S)))$
- Merge phase reads each relation once
- Total IO
 - $b(R) + b(S) + 2*b(R)*\text{ceil}(\log_M(b(R))) + 2*b(S)*\text{ceil}(\log_M(b(S)))$

This is only the case when, for each join-value b , the R and S blocks with that value b fit together in the buffer.

If this is not the case for value b , then the sort-merge algorithm needs to do a local block-nested join loop on the R and S blocks with value b .

Better than Block-Nested-Loop?

- Assume $b(R)=10,000$, $b(S)=2,000$, $M=500$
 - BNL costs 42,000
 - With S as outer relation
 - SM: $10,000+2,000+4*10,000+4*2,000 = 60,000$ since
 $\text{ceil}(\log_{500}(10,000)) = \text{ceil}(\log_{500}(2,000)) = 2$
- Assume $b(R)=1,000.000$, $b(S)=1,000$, $M=500$
 - BNL costs $1,000 + 1,000,000*1000/500 = 2,001,000$
 - SM: $1.000.000+1.000+6*1.000.000+4*1.000 = 7,005,000$

Comparison

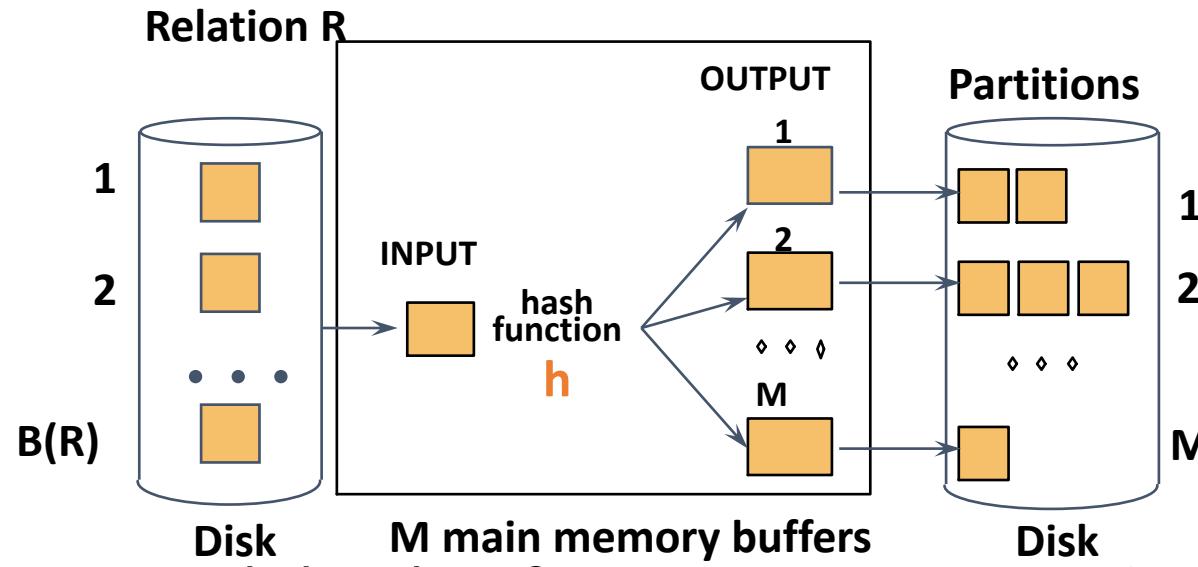
- Assume relations of equal size: B blocks
- SM: $2*B + 4*B*\log_M(B)$
- BNL: $B+B^2/M$
- BNL > SM
 - $B+B^2/M > 2*B + 4*B*\log_M(B)$
 - $B/M > 1+ 4*\log_M(B)$
 - $B > M + 4*M*\log_M(B)$

Hash Join

- As always, we may save sorting if good hash function available
- Assume a **very good** hash function
 - Distributes hash values **almost uniformly** over hash table
 - If we have **good histograms** (later), a simple interval- based hash function will usually work
- How can we apply hashing to joins?

Hashing a file on join attribute(s)

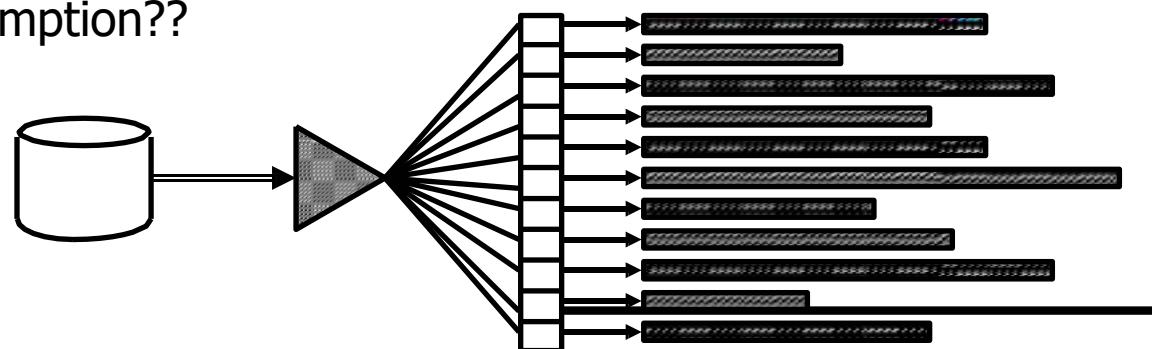
- Idea: partition a relation R into buckets, on disk
- Each bucket has size approx. $B(R)/M$



- Does each bucket fit in main memory ?
 - Yes if $B(R)/M \leq M$, i.e. $B(R) \leq M^2$

Hash Join Idea

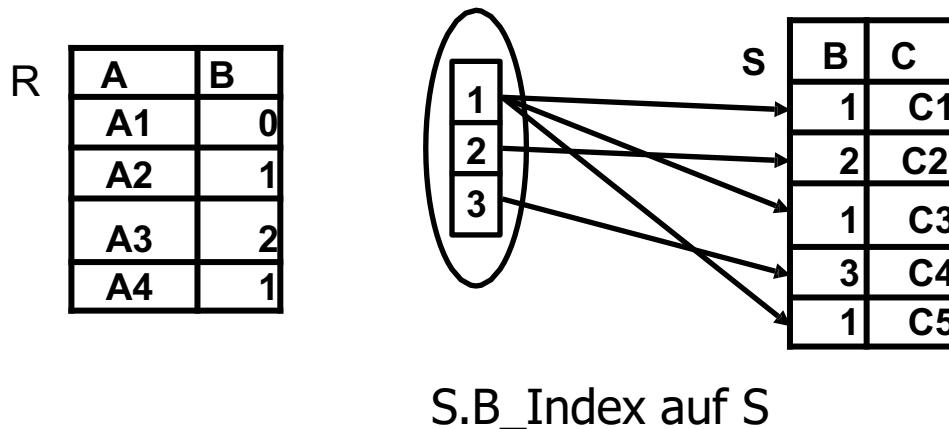
- Use join attributes as hash keys in both R and S
- Choose hash function for **hash table of size M**
 - Each bucket has size $b(R)/M$, $b(S)/M$
- Hash phase
 - Scan R, compute hash table, **writing full blocks to disk immediately**
 - Scan S, compute hash table, **writing full blocks to disk immediately**
- Merge phase
 - Iteratively, load same bucket of R and of S in memory
 - Compute join
- Total cost
 - **Hash phase** costs $2*b(R)+2*b(S)$
 - **Merge phase** costs $b(R) + b(S)$
 - Total: **$3*(b(R)+b(S))$**
 - Under what assumption??



Index Join

- Assume we have an index “`B_Index`” on one join attribute
- Choose indexed relation as inner relation
- **Index join**

```
FOR EACH r IN R DO
    X = { SEARCH (S.B_Index, <r.B>) }
    FOR EACH TID i in X DO
        s = READ (S, i) ; output (r ⚡ s).
```



- Actually, this is a one **block-nested loop with index access**
 - Using BNL possible (and better)

Semi Join

- Consider queries such as
 - `SELECT DISTINCT R.* FROM S,R WHERE R.B=S.B`
 - `SELECT R.* FROM R WHERE R.B IN (SELECT S.B FROM S)`
- What's special?
 - No values from S are requested in result
 - S (or inner query) acts as filter on R
- Semi-Join $R \ltimes S$

Implementing Semi-Join

- Using blocked-nested-loop join
 - Choose relation R as outer relation
 - Perform BNL
 - Whenever partner for R.B is found, exit inner loop
- Using sort-merge join
 - Sort R
 - Sort join attribute values from S, remove duplicates on the way
 - Perform merge phase as usual
- Using hash join
 - Hash R
 - Hash join values from S, remove duplicates on the way
 - Perform hash phase as usual

Implementing Intersection, Union, Difference

- Analogous with semi-join
- Using sort-merge join
 - Sort R, remove duplicates on the way
 - Sort S, remove duplicates on the way
 - Perform merge phase as usual (checking for and, or, not)
- Using hash join
 - Hash R, remove duplicates on the way
 - Hash S, remove duplicates on the way
 - Perform hash phase as usual (checking for and, or, not)

Observe that the performance is never $O(R*S)$