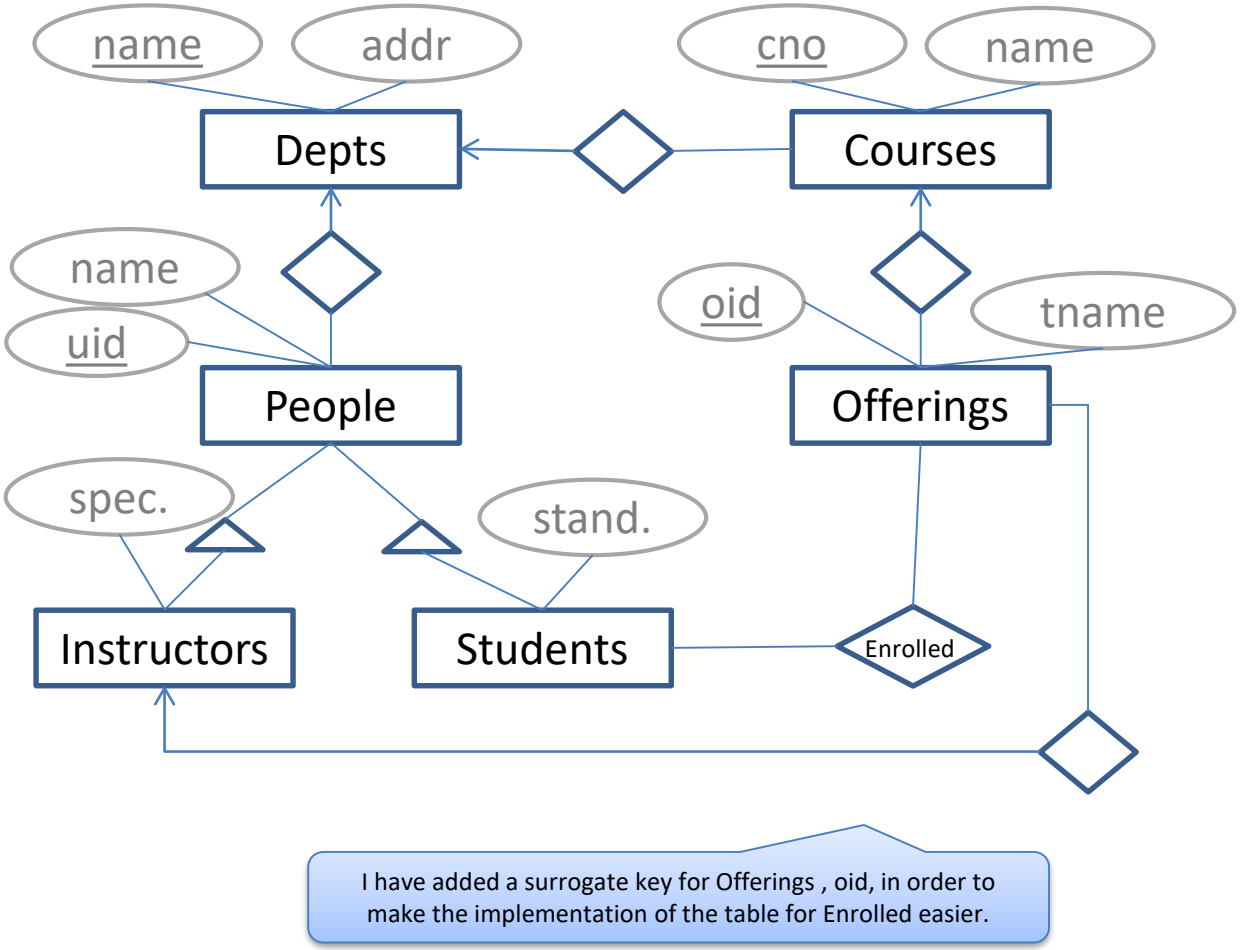
Review

ER, Tables, Queries, Constraints



I have added a surrogate key for Offerings , oid, in order to make the implementation of the table for Enrolled easier.

```
CREATE TABLE Depts (
  name VARCHAR(20) PRIMARY KEY,
  addr VARCHAR(20)
);

CREATE TABLE People (
  uid CHAR(10) PRIMARY KEY,
  name VARCHAR(20),
  dname VARCHAR(20) REFERENCES Depts(name)
);

CREATE TABLE Instructors (
  uid CHAR(10) PRIMARY KEY REFERENCES People(uid),
  specialty VARCHAR(40)
);

CREATE TABLE Students (
  uid CHAR(10) PRIMARY KEY REFERENCES People(uid),
  standing VARCHAR(30)
);

CREATE TABLE Courses (
  cno CHAR(8) PRIMARY KEY,
  name VARCHAR(40),
  dname VARCHAR(20) REFERENCES Depts(name)
);

CREATE TABLE Offerings (
  oid INT PRIMARY KEY,
  cno CHAR(8) REFERENCES Courses(cno),
  tname CHAR(6),
  uid CHAR(10) REFERENCES Instructors(uid)
);

CREATE TABLE Enrolled (
  oid INT REFERENCES Offerings(oid),
  uid CHAR(10) REFERENCES Students(uid),
  PRIMARY KEY(oid,uid)
);
```

```
INSERT INTO Depts VALUES('Comp.Sci.', 'ECS Bldg');
INSERT INTO People VALUES('V111111111', 'Jon', 'Comp.Sci.');
INSERT INTO People VALUES('V222222222', 'Ben', 'Comp.Sci.');
INSERT INTO Instructors VALUES('V111111111', 'Hardware');
INSERT INTO Students VALUES('V222222222', '3rd year');
INSERT INTO Courses VALUES('CSC390', 'Hardware Systems', 'Comp.Sci.');
INSERT INTO Offerings VALUES(1, 'CSC390', '201409', 'V111111111');
INSERT INTO Enrolled VALUES(1, 'V222222222');
```

Exercise – mutually exclusive subclasses

```
CREATE TABLE Vehicles (
  vin CHAR(17) PRIMARY KEY,
  vehicle_type CHAR(3) CHECK(vehicle_type IN ('SUV', 'ATV')),
  fuel_type CHAR(4),
  door_count INT CHECK(door_count >= 0),
  UNIQUE(vin, vehicle_type)
);

CREATE TABLE SUVs (
  vin CHAR(17) PRIMARY KEY,
  vehicle_type CHAR(3) CHECK(vehicle_type ='SUV'),
  FOREIGN KEY (vin, vehicle_type) REFERENCES Vehicles (vin, vehicle_type)
);

CREATE TABLE ATVs (
  vin CHAR(17) PRIMARY KEY,
  vehicle_type CHAR(3) CHECK(vehicle_type ='ATV'),
  FOREIGN KEY (vin, vehicle_type) REFERENCES Vehicles (vin, vehicle_type)
);
```

Views with check option: Example

```
CREATE TABLE Hotel (
  room_nbr INT,
  arrival_date DATE,
  departure_date DATE,
  guest_name CHAR(15),
  PRIMARY KEY (room_nbr, arrival_date),
  CHECK (departure_date > arrival_date)
);
```

We want to add the constraint that reservations do not overlap.

```
CREATE VIEW HotelStays AS
    SELECT room_nbr, arrival_date, departure_date, guest_name
    FROM Hotel H1
    WHERE NOT EXISTS (
      SELECT *
      FROM Hotel H2
      WHERE H1.room_nbr = H2.room_nbr AND
          (H2.arrival_date < H1.arrival_date AND H1.arrival_date < H2.departure_date)
    )
    WITH CHECK OPTION;
```

SQL

SELECT … FROM … WHERE … GROUP BY … HAVING … ORDER BY

JOIN ON
JOIN USING
LEFT JOIN
RIGHT JOIN

LIKE

IS NULL
IS NOT NULL

Careful what can go to SELECT when having GROUP BY

Careful regarding the difference in conditions that go to HAVING vs those that go to WHERE.

Example

Canadian stars
who first appeared in
a movie before 2000

```
SELECT si.starName, SUM(length)
FROM Movies m
        JOIN StarsIn si USING(title,year)
        JOIN Stars s ON s.name=si.starName
WHERE s.birthplace LIKE '%Canada%'
GROUP BY si.starName
HAVING MIN(si.year) < 2000;
```

Correlated Subqueries

- Suppose StarsIn table has an additional attribute “salary”

  StarsIn(movie, movie, starName, salary)

  Now, find the stars who were paid for some movie more than the average salary for that movie.

```
SELECT starName, title, year
FROM StarsIn X
WHERE salary >
        (SELECT AVG(salary)
        FROM StarsIn
        WHERE title = X.title AND year=X.year);
```

Remark

Semantically, the value of the X tuple changes in the outer query, so the database must rerun the subquery for each X tuple.

Another Solution (Nesting in FROM)

```
SELECT X.starName, X.title, X.year
    FROM StarsIn X, (SELECT title, year, AVG(salary) AS avgSalary
                     FROM StarsIn
                     GROUP BY title, year) Y
    WHERE X.salary>Y.avgSalary AND
                  X.title=Y.title AND X.year=Y.year;
```

orders per **month** per **category**

```
CREATE TABLE T AS
SELECT
        TO_CHAR(orderdate, 'YYYY') AS year,
        TO_CHAR(orderdate, 'MM') AS month,
        productgroupname AS cat,
        COUNT(orderid) AS countorders,
        SUM(orderline.totalprice) AS revenue
FROM orders JOIN
          orderline USING(orderid) JOIN
          products USING(productid)
GROUP BY
      TO_CHAR(orderdate, 'YYYY'),
      TO_CHAR(orderdate, 'MM'),
      productgroupname
ORDER BY 1,2;
```

year	month	cat	countorders	revenue
2009	10	ARTWORK	1782	45416
2009	10	BOOK	731	15299
2009	10	OCCASION	169	3476
2009	11	ARTWORK	2138	79390
2009	11	BOOK	2353	45808
2009	11	OCCASION	485	10041
2009	12	APPAREL	17	719
…	…	…	…	…

First: Mix **detail** and aggregate information over **window**

```
SELECT cat, year, month, revenue,
       AVG(revenue) OVER (PARTITION BY cat, year) AS avgrev
FROM T;
```

Detail

Window
All tuples of **T** with same **cat** and **year** as in the detail part.

cat	year	month	revenue	avgrev
APPAREL	2009	12	719.1	719.1
ARTWORK	2009	12	32924.25	52576.42
ARTWORK	2009	10	45415.5	52576.42
ARTWORK	2009	11	79389.5	52576.42
BOOK	2009	11	45808.31	26678
BOOK	2009	12	18926.84	26678
…				

Second: Extract what you want with enclosing query

```
SELECT cat, year, month
FROM
   (SELECT cat, year, month, revenue,
          AVG(revenue) OVER (PARTITION BY cat, year) AS avgrev
    FROM T) X
WHERE revenue < avgrev
ORDER BY cat, year, month;
```

cat	year	month	revenue	avgrev
APPAREL	2009	12	719.1	719.1
ARTWORK	2009	12	32924.25	52576.42
ARTWORK	2009	10	45415.5	52576.42
ARTWORK	2009	11	79389.5	52576.42
BOOK	2009	11	45808.31	26678
BOOK	2009	12	18926.84	26678
…				

Which months did revenues from a category drop below those of the same month one year ago without increasing again the next year?

```
SELECT *
FROM (
  SELECT year, month, cat, revenue,
        LAG(revenue,12) OVER (PARTITION BY cat ORDER BY year, month)
                                                    AS prev_year_rev,
        LEAD(revenue,12) OVER (PARTITION BY cat ORDER BY year, month)
                                                    AS next_year_rev
  FROM T) X
WHERE revenue < prev_year_rev AND next_year_rev <= revenue
ORDER BY year, month;
```

year	month	cat	revenue	prev_year_rev	next_year_rev
2011	5	ARTWORK	64538.66	64620.75	32268.12
2011	7	ARTWORK	49170.82	61181.51	42955.31
2011	9	OTHER	887.8	1939.79	364.47
2011	10	OTHER	1000.21	3154.7	297.26
2011	11	BOOK	56548.76	62757.74	31030.95
…					

Which are the top 10 months in terms of revenue for each category?

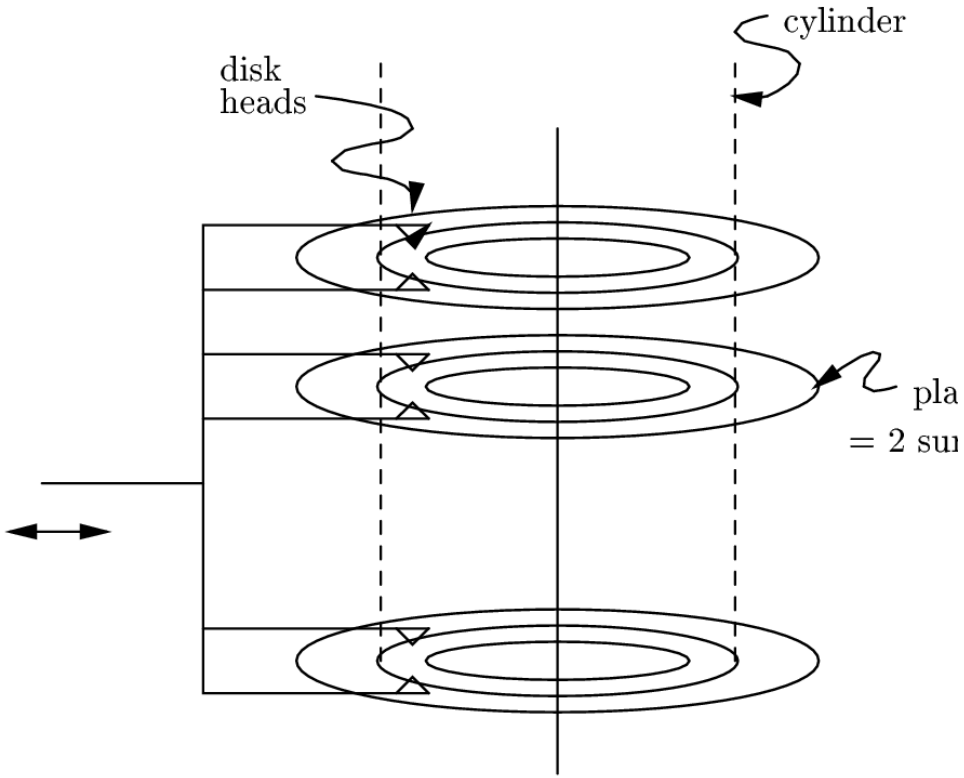
ROW_NUMBER is a binary operator; it takes a tuple and an ordered set and returns the rank of the tuple in the set. It is always for sure that the tuple is member of the set.

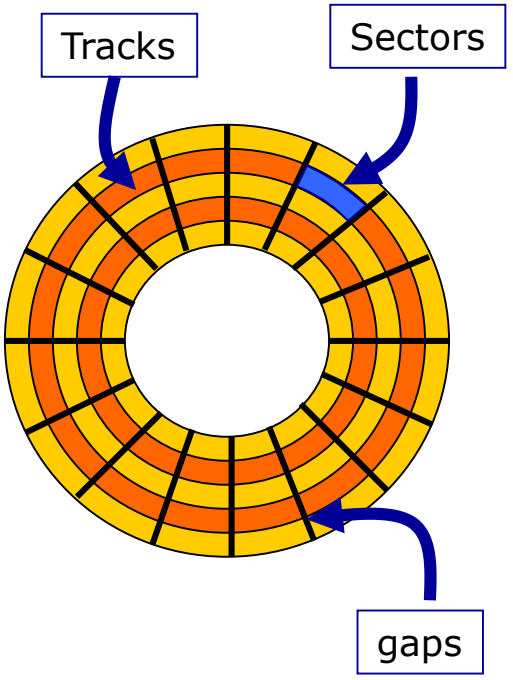
```
SELECT *
FROM (
  SELECT cat, year, month, round(revenue,-3),
     ROW_NUMBER() OVER (PARTITION BY cat ORDER BY round(revenue,-3) DESC) AS rank
  FROM T) X
WHERE rank<=10
ORDER BY cat, rank;
```

- The “window” is the subset of tuples with same **cat** as the detail (first part of SELECT).
- The window is ordered by revenue (rounded to the nearest thousand). Ties are broken arbitrarily.
- ROW_NUMBER() returns the rank of the detail in the ordered window.

External Storage

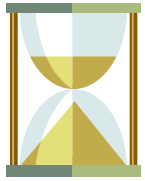
Disks





Disk access time

- **Seek time** = time to move heads to proper cylinder (track).
- **Rotational delay** = time for desired block to come under head.
- **Transfer time** = time during which the block passes under head.



MIN time to read a 16,384-byte block

- The minimum time, is just the transfer time.
- Heads must pass over 4 sectors and the 3 gaps between them.
- Gaps represent 10% of the circle and sectors the remaining 90%,
  - 36 degrees are occupied by gaps and
  - 324 degrees by the sectors.
  - 256 gaps and 256 sectors around the circle,
- So
  - a gap is 36/256 = 0.14 degrees, and
  - a sector is 324/256 = 1.265 degrees
- Total degrees covered by 3 gaps and 4 sectors is:
  - 3*0.14+4*1.265 = 5.48 degrees
- Transfer time: (5.48/360) x 8.33 = **0.13 ms**

MAX time to read a 16,384-byte block

- Worst-case latency is 17.38+8.33+0.13=**25.84 ms**.

Time to travel 65,536 cylinders at 4000 cyls/ms plus 1ms for start-stop

Time for a full rotation.

There are 7200 rotations per min. So, 1/7200 is the time in **min** for one rotation.

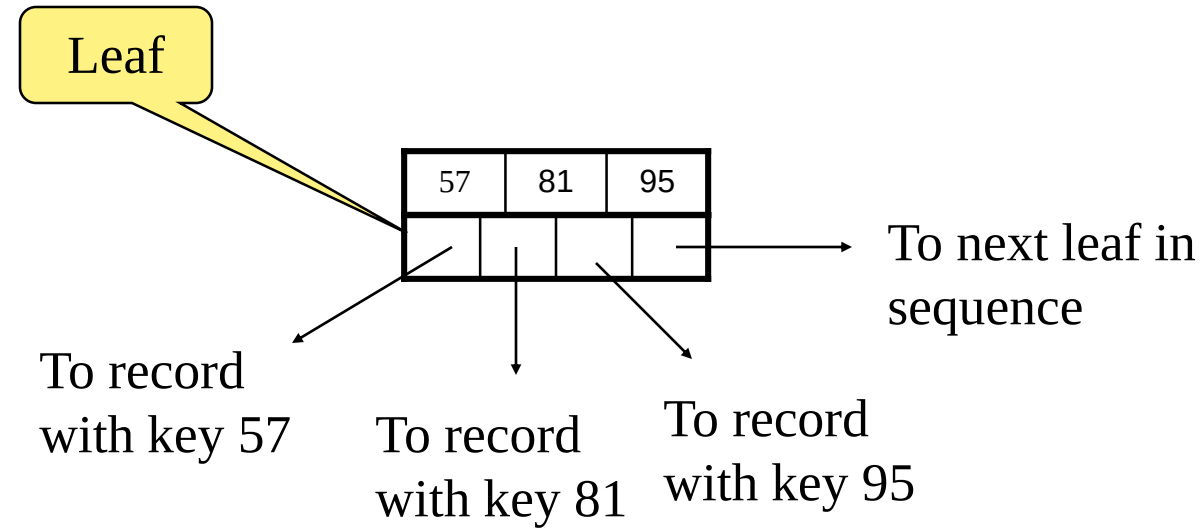
60*1000/7200 = 8.33ms is time in **ms** for one rotation.

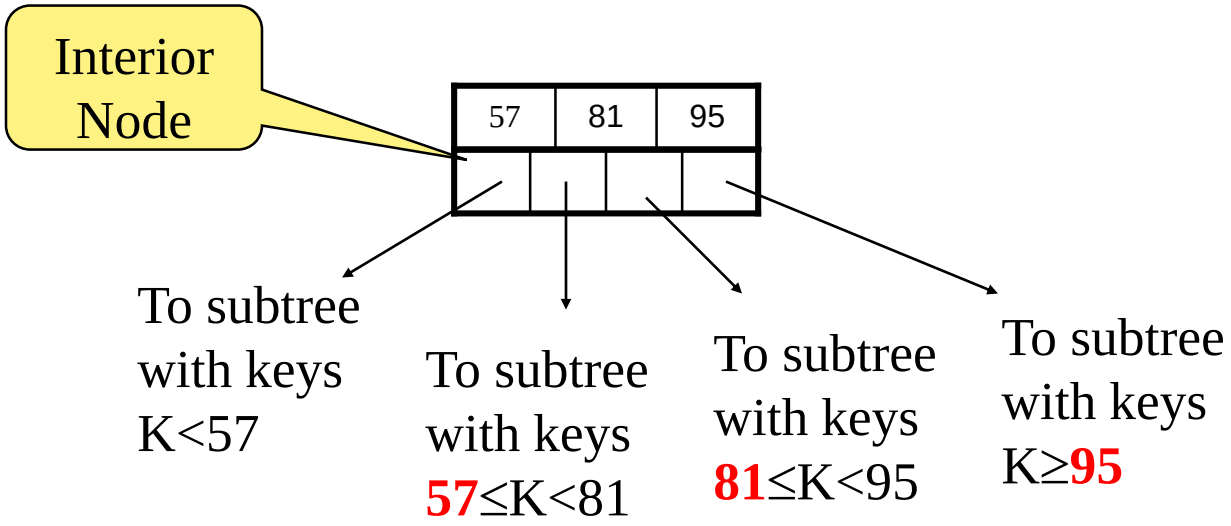
AVG time to read a 16,384-byte block

- Transfer time is **0.13** milliseconds
- Average rotational delay is the time to rotate the disk half way around, or **4.17** milliseconds.
- Average seek time is: 1+(65536/3)*(1/4000) = **6.46** ms
- Total: 6.46 + 4.17 + 0.13 = 10.76 ms (or about 11 ms)

Indexes

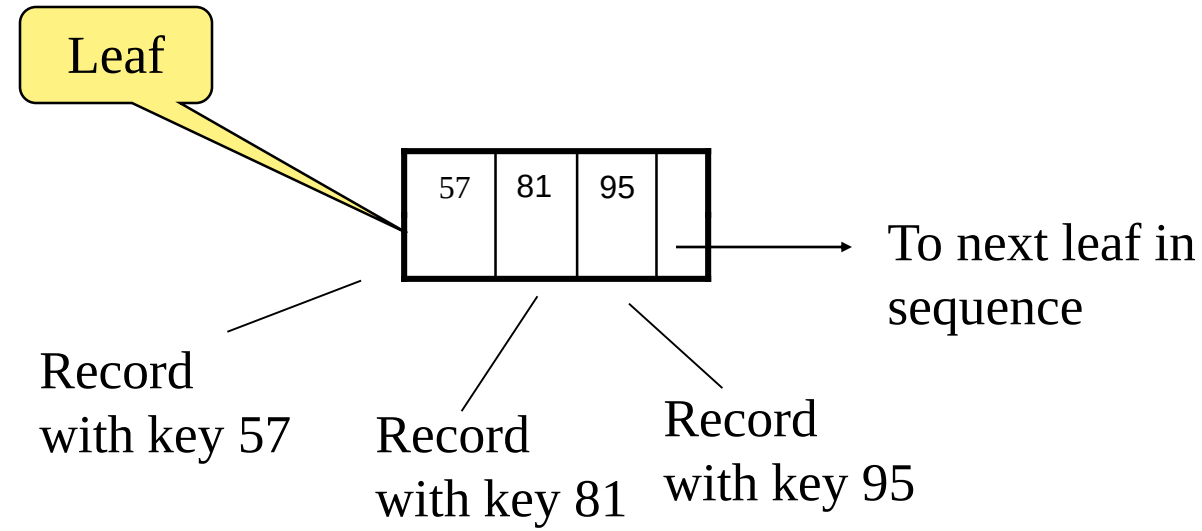
BTrees: Leaf and interior node (unclustered)

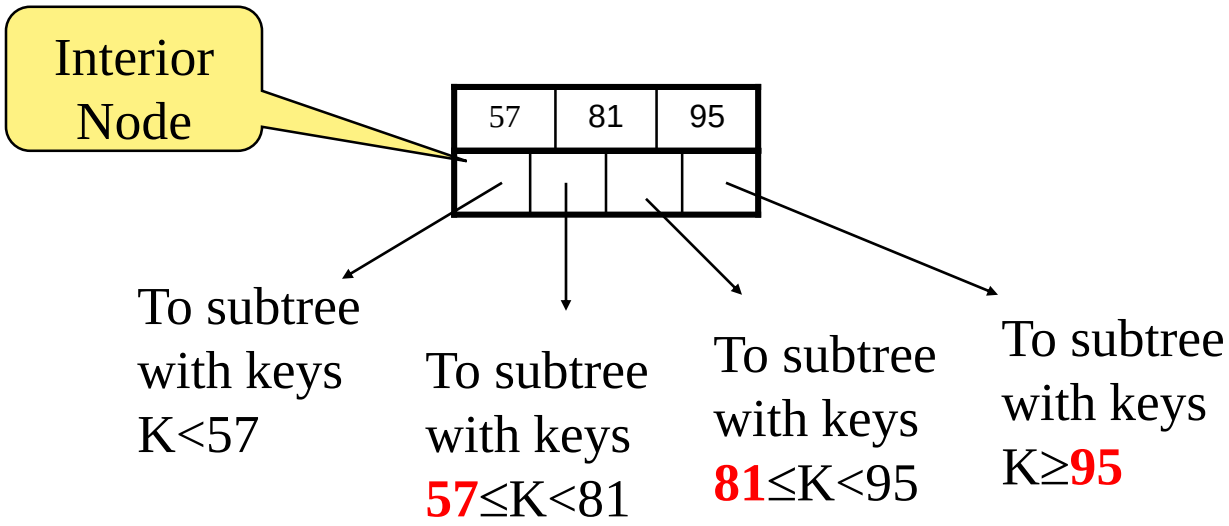




57, **81**, and **95** are **the least keys** we can reach by via the corresponding pointers.

BTrees: Leaf and interior node (clustered)





57, **81**, and **95** are **the least keys** we can reach by via the corresponding pointers.

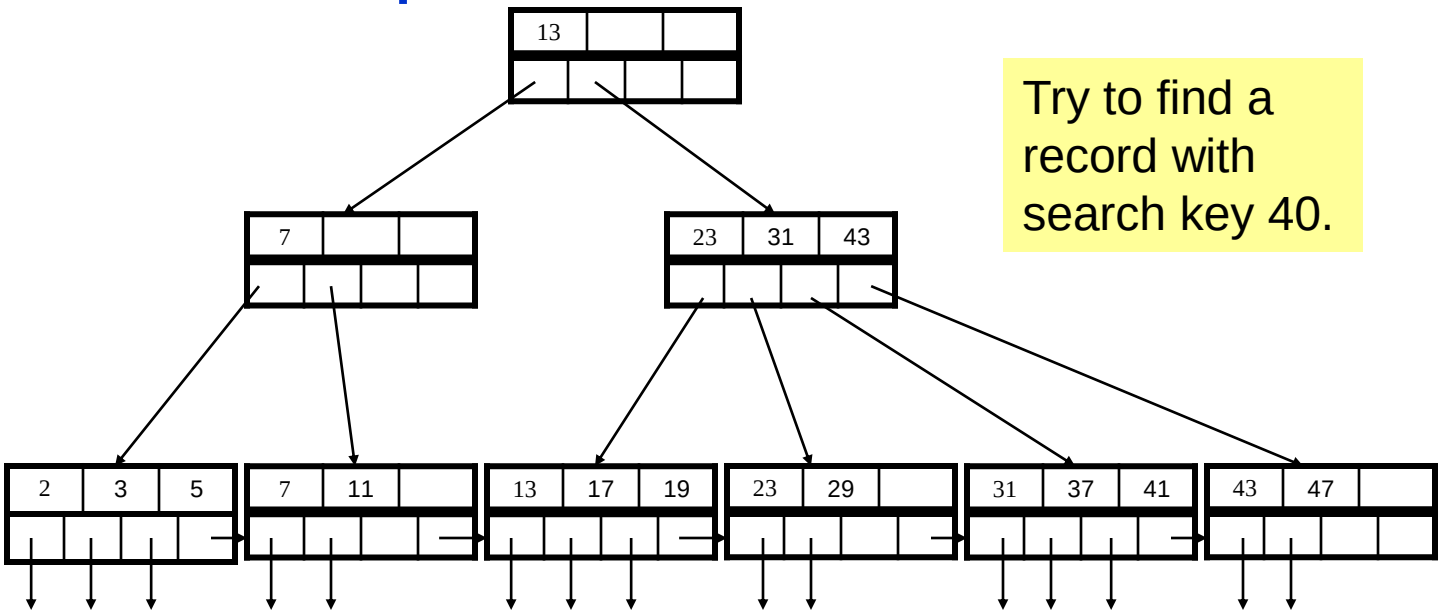
Operations in B-Tree

Will illustrate with unclustered case, but straightforward to generalize for the clustered case.

Operations

1. Lookup
2. Insertion
3. Deletion (optional material)

Lookup



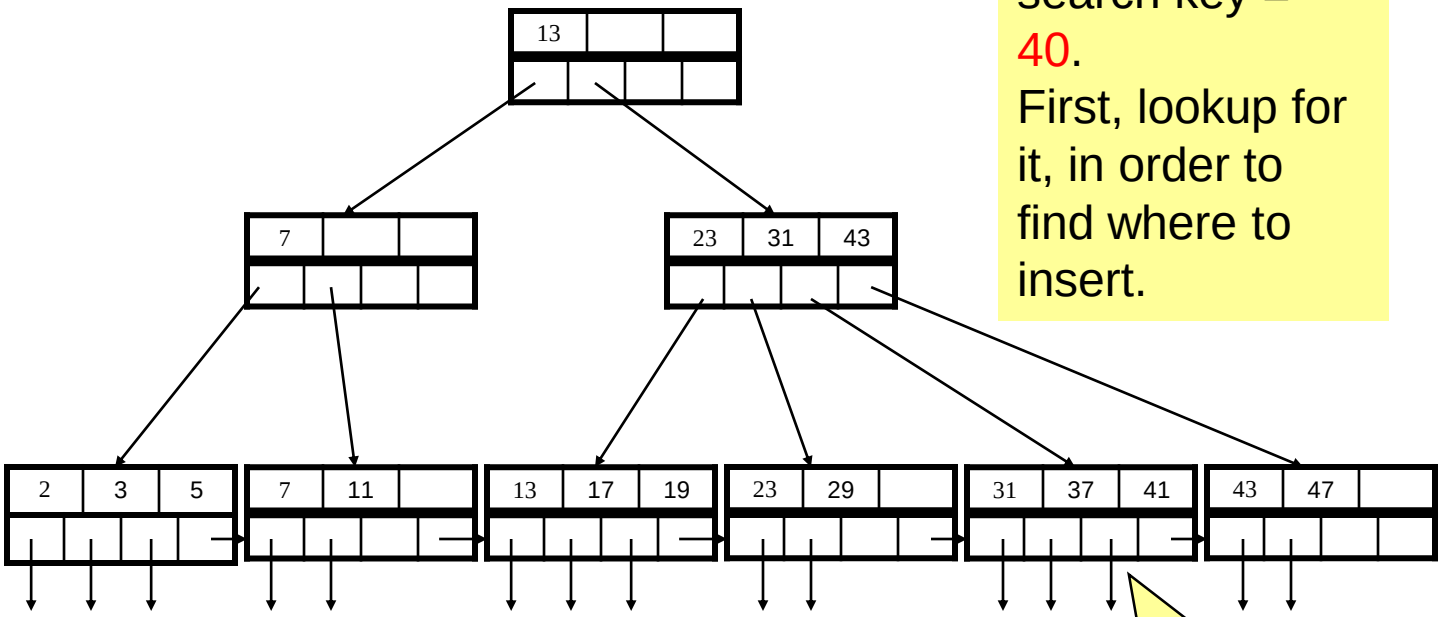
Try to find a record with search key 40.

Recursive procedure:

If we are at a leaf, look among the keys there. If the *i*-th key is K, the the *i*-th pointer will take us to the desired record.

If we are at an internal node with keys $K_1,K_2,\ldots,K_n$, then if $K<K_1$ we follow the first pointer, if $K_1 \leq K < K_2$ we follow the second pointer, and so on.

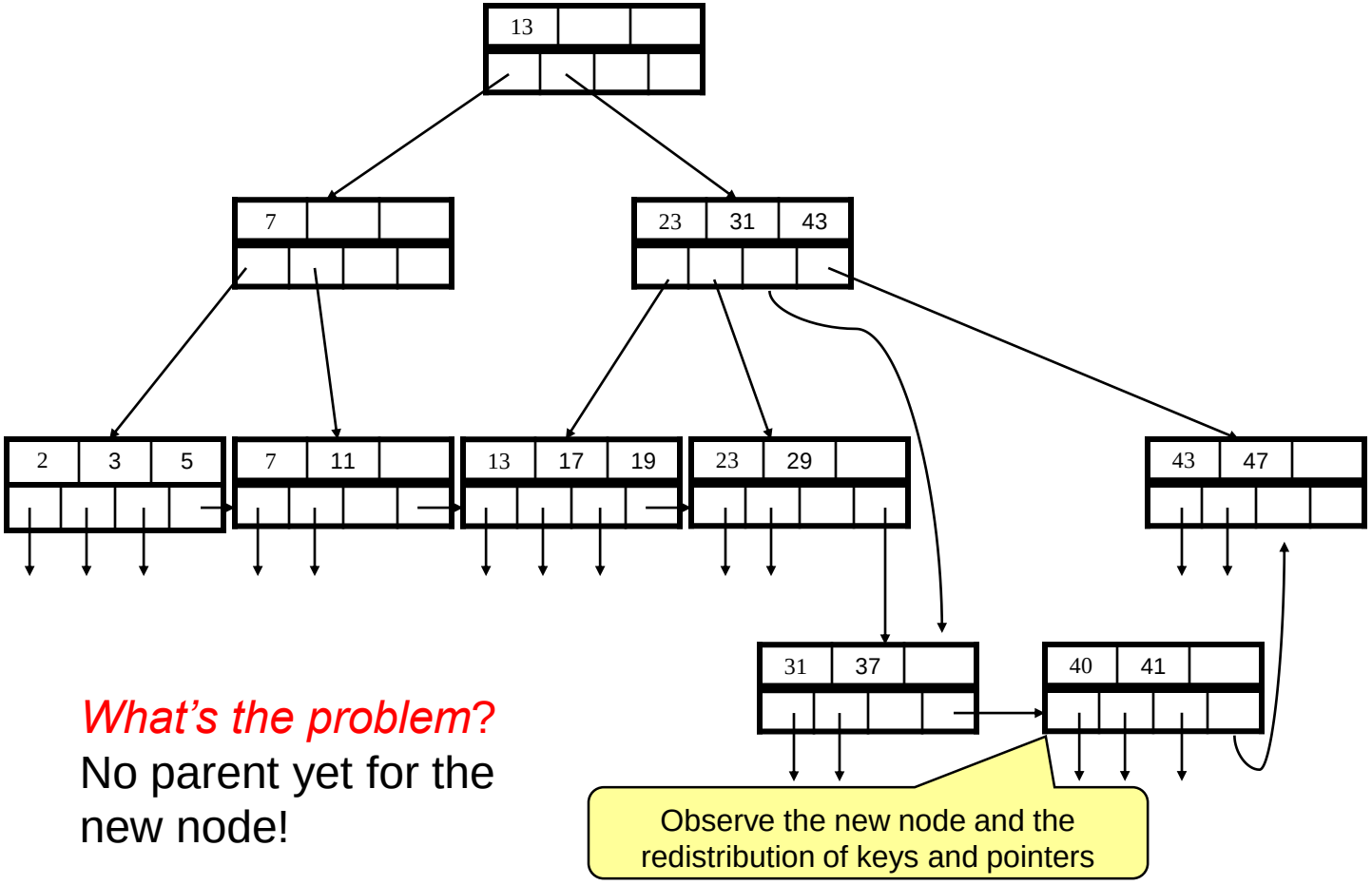
Insertion



Try to insert a search key = 40.
First, lookup for it, in order to find where to insert.

It has to go here, but the node is full!

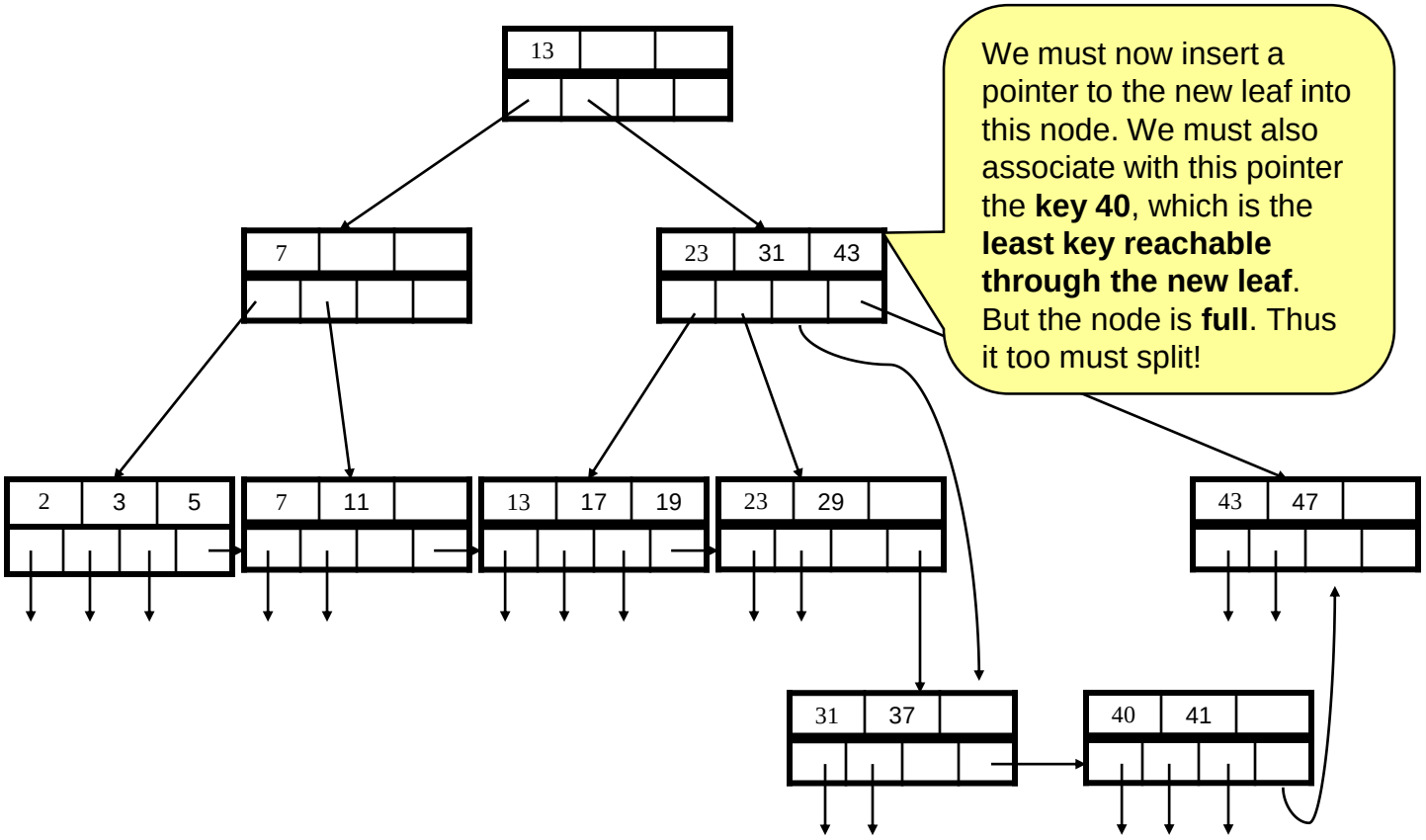
Beginning of the insertion of key 40



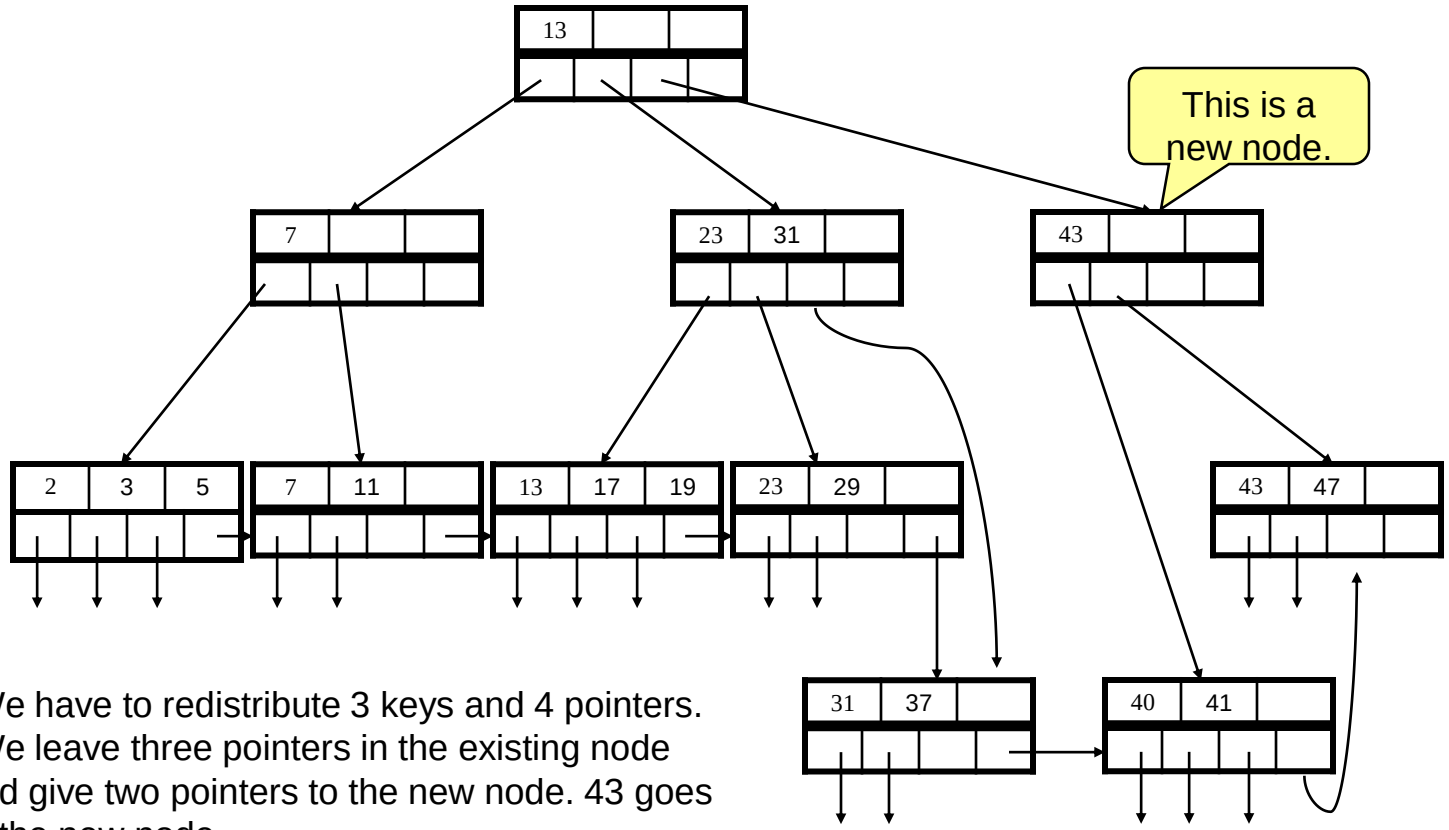
What's the problem?

No parent yet for the new node!

Continuing of the Insertion of key 40

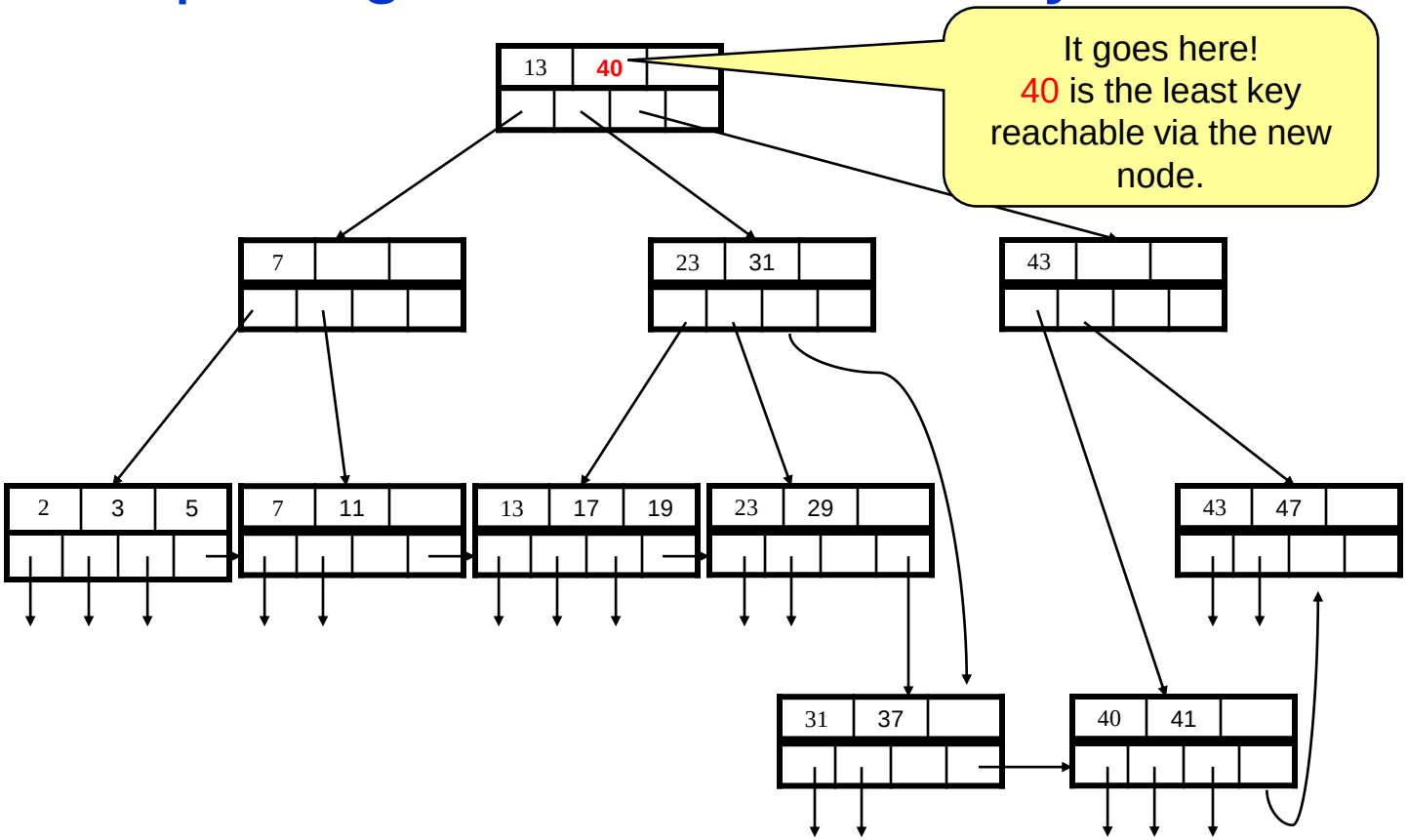


Completing of the Insertion of key 40



•We have to redistribute 3 keys and 4 pointers.
•We leave three pointers in the existing node and give two pointers to the new node. 43 goes to the new node.
•But where does the key 40 goes?
•40 is the least key reachable via the new node.

Completing the Insertion of key 40



Structure of B-trees with real blocks

- Degree n means that all nodes have space for n search keys and $n+1$ pointers
- **Node = block**
- Let
  - block size be 16,384 Bytes,
  - key 20 Bytes,
  - pointer 20 Bytes.
- Let's solve for n:

$20n + 20(n+1) \leq 16384$

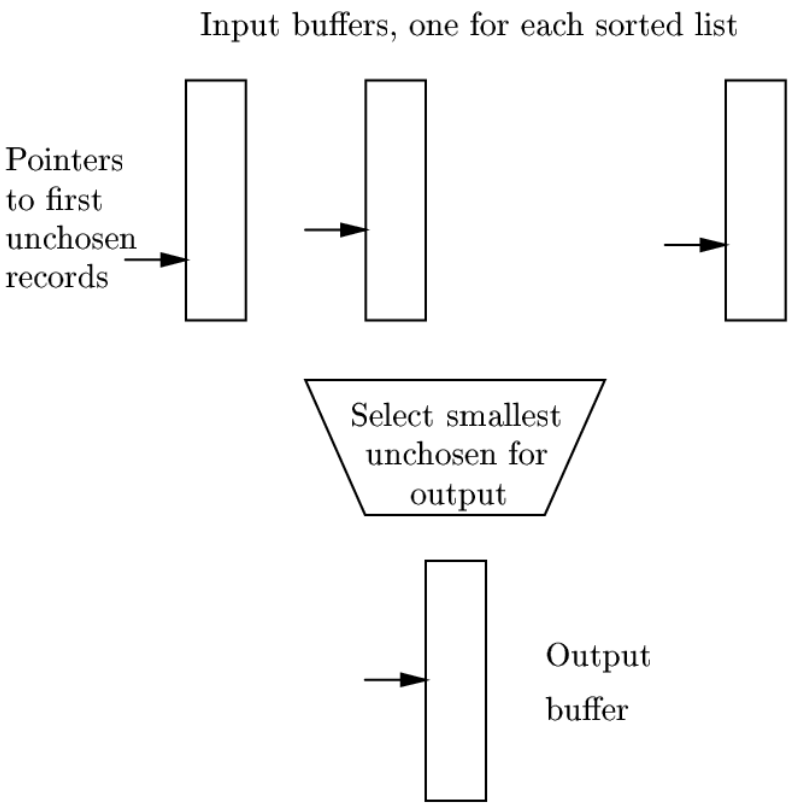
$\Rightarrow \; n \leq 409$

2PMMS

Phase 1

1. Fill main memory with records.
2. Sort using favorite main memory sort.
3. Write sorted sublist to disk.
4. Repeat until all records have been put into one of the sorted lists.

Phase 2



How many records can we sort?

1. Block size is B bytes.
2. Main memory available for buffering blocks is M bytes.
3. Record is R bytes.

- Number of main memory buffers = M/B blocks
- We need one output buffer, so we can actually use (M/B)-1 input buffers.

- How many sorted sublists can we merge?
- (M/B)-1.

- What's the total number of records we can sort?
- Each time we fill in the memory with M/R records.
- Hence, we are able to sort (M/R)*[(M/B)-1] or approximately M^2/RB.

If we use the parameters in the example about 2PMMS we have:

M=100MB = 100,000,000 Bytes = 10^8 Bytes ($100*2^{20}$ more precisely)

B = 16,384 Bytes

R = 160 Bytes

So, M^2/RB = (100*2^20)^2 / (160 * 16,384) = 4.2 billion records, or 2/3 of a TeraByte.

Sorting larger files

- If our file is bigger, then, we can use 2PMMS to create sorted sublists of **M^2/RB** records.
- Then, in a third pass, we can merge **$(M/B)-1$** of these sorted sublists.
- Thus, the third phase let's us sort
  - **$[(M/B)-1]*[M^2/RB] \approx M^3/RB^2$** records

For our example, the third phase let's us sort 75 trillion records occupying 7500 Petabytes!!

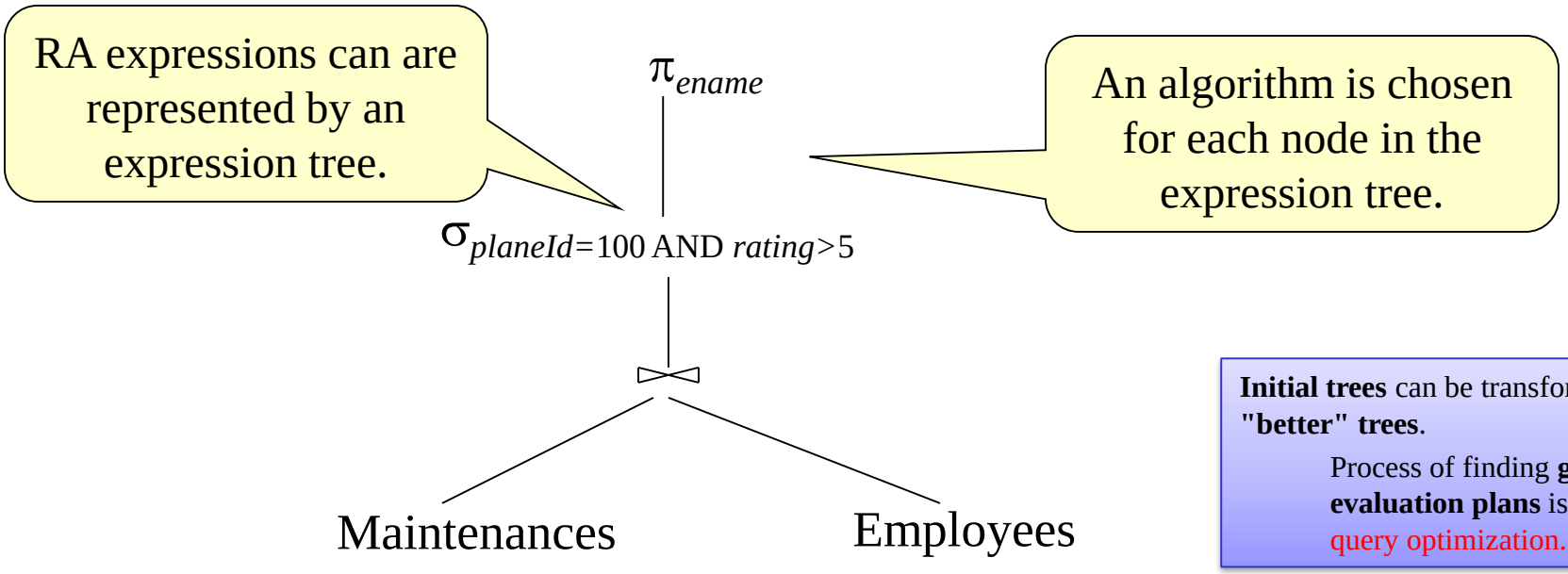
Query Evaluation

An SQL query and its RA equiv.

Employees (sin INT, ename VARCHAR(20), rating INT, address VARCHAR(90))
Maintenances (sin INT, planeId INT, day DATE, desc CHAR(120))

```
SELECT ename
FROM Employees NATURAL JOIN Maintenances
WHERE planeId = 100 AND rating > 5;
```

$\pi_{ename}(\sigma_{planeId=100 \text{ AND } rating>5}(\text{Employees} \bowtie \text{Maintenances}))$



Initial trees can be transformed to **"better" trees**.

Process of finding **good evaluation plans** is called query optimization.

Running Example – Airline

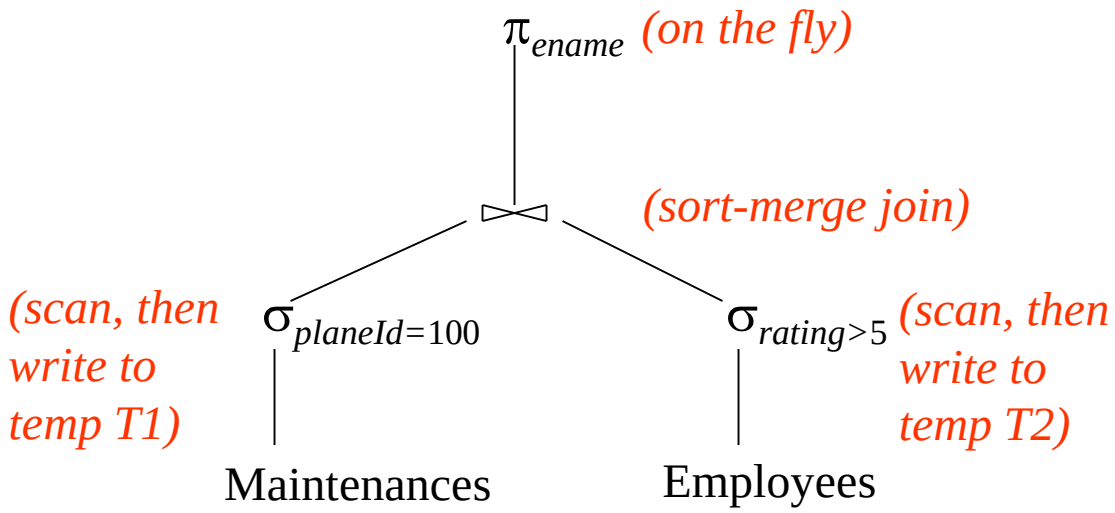
Employees (sin INT, ename VARCHAR(20), rating INT, address VARCHAR(90))

Maintenances (sin INT, planeId INT, day DATE, desc CHAR(120))

- Assume for **Maintenances**:
  - a tuple is 160 bytes
  - a block can hold 100 tuples (16K block)
  - we have **1000 blocks** of such tuples.

- Assume for **Employees**:
  - a tuple is 130 bytes
  - a block can hold 120 tuples
  - we have **50 blocks** of such tuples.

Cost of a Plan (Pushing selections)



(1000+10) + (50+25) + (4*10 + 4*25) + (10+25) = **1260** I/Os

1000 I/Os to scan Maintenances,
10 I/Os to save temporary table T1 (assuming there are 100 planes and for each we have an equal number of maintenance records)

50 I/Os to scan Employees,
25 I/Os to save temporary table T2 (assuming there are 10 ratings uniformly distributed)

Cost for sorting T1 and T2 on SIN (the join attribute) using 2PMMS.

Cost for reading the sorted T1 and T2 and doing the join.

See the full slides more examples

Concurrency

Summarizing the Terminology

- **Transaction:** *sequence* of *r* and *w* actions on database elements.
- **Schedule:** *sequence* of read/write actions performed by a collection of transactions.
- **Serial Schedule**: all actions for each transaction are consecutive.

  r1(A); w1(A); r1(B); w1(B); r2(A); w2(A); r2(B); w2(B); …
- **Serializable Schedule**: schedule whose "**effect**" is equivalent to that of some serial schedule.
- **Conflict-serializable Schedule:** can be converted into a serializable schedule with the **same effect** by a series of **non-conflicting swaps** of adjacent elements

Example of a conflict-serializable schedule

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_2(A); \mathbf{w_2(A)}; \mathbf{r_1(B)}; w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); \mathbf{r_2(A)}; \mathbf{r_1(B)}; w_2(A); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); r_2(A); \mathbf{w_2(A)}; \mathbf{w_1(B)}; r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); \mathbf{r_2(A)}; \mathbf{w_1(B)}; w_2(A); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A) w_2(A); r_2(B); w_2(B)$

The operations in bold can be safely swapped.

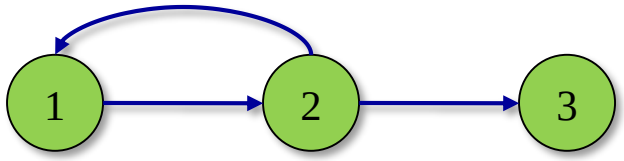
Precedence graphs

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$

Note the following:

- $r_2(B) <_S w_1(B)$
- $w_2(A) <_S w_3(A)$
- $r_1(B) <_S w_2(B)$

➢ Here, we have $T_1 < T_2 < T_3$, but we also have $T_2 < T_1$



Not conflict serializable

Two Phase Locking

Simple condition, which guarantees conflict-serializability:

In every transaction, all lock requests (phase 1) precede all unlock requests (phase 2).

T_1	T_2	A	B
		25	25
$l_1(A)$; $r_1(A)$			
A = A + 100			
$w_1(A)$; $l_1(B)$; $u_1(A)$		125	
	$l_2(A)$; $r_2(A)$		
	A = A * 2		
	$w_2(A)$	250	
	$l_2(B)$ **Denied**		
$r_1(B)$			
B = B + 100			125
$w_1(B)$; $u_1(B)$			
	$l_2(B)$; $u_2(A)$; $r_2(B)$		
	B = B * 2		
	$w_2(B)$;$u_2(B)$		250

Shared/Exclusive Locks

	S	X
S	yes	no
X	no	no

Shared/Exclusive Locks Example

$r_1(A); r_2(B); r_3(C); r_1(B); r_2(C); r_3(D); w_1(A); w_2(B); w_3(C);$

T_1	T_2	T_3
$xl_1(A); r_1(A)$		
	$xl_2(B); r_2(B)$	
		$xl_3(C); r_3(C)$
$sl_1(B)$ **denied**		
	$sl_2(C)$ **denied**	
		$sl_3(D); r_3(D);$
		$w_3(C); u_3(C); u_3(D)$
	$sl_2(C); r_2(C);$	
	$w_2(B);$	
	$u_2(B); u_2(C)$	
$sl_1(B); r_1(B);$		
$w_1(A);$		
$u_1(A); u_1(B)$		

If a transaction T_i only reads element X, it requests $sl_i(X)$.

However, if T_i reads and then writes X, it only requests $xl_i(X)$.

This example shows that when there are transactions that will eventually write the elements they read, having shared and exclusive locks isn't much better than having just simple locks.

Upgrading Locks

$r_1(A); r_2(B); r_3(C); r_1(B); r_2(C); r_3(D); w_1(A); w_2(B); w_3(C);$

T_1	T_2	T_3
$sl_1(A); r_1(A)$		
	$sl_2(B); r_2(B)$	
		$sl_3(C); r_3(C)$
$sl_1(B); r_1(B)$		
	$sl_2(C); r_2(C)$	
		$sl_3(D); r_3(D)$
$xl_1(A); w_1(A);$		
$u_1(A); u_1(B)$		
	$xl_2(B); w_2(B);$	
	$u_2(B); u_2(C)$	
		$xl_3(C); w_3(C);$
		$u_3(C); u_3(D)$

Now, with locks that can be upgraded, transactions ask first for shared locks to read elements, then ask for the locks to be upgraded to exclusive when they want to write those elements.

Everything goes smoothly without any request being denied.

Possibility for Deadlocks

Example:T1 and T2 each reads X and later writes X.

T_1	T_2
$sl_1(X)$	
	$sl_2(X)$
$xl_1(X)$ Denied	
	$xl_2(X)$ Denied

Problem: when we allow upgrades, it is easy to get into a deadlock situation.

Solution: Update Locks

- Update lock **$ul_i(X)$**.
  - Only an update lock (not shared lock) can be upgraded to exclusive lock
  - A transaction that will read and later on write some element A, asks initially for an update lock on A, and then asks for an exclusive lock on A. Such transaction doesn't ask for a shared lock on A.

	S	X	U
S	yes	no	yes
X	no	no	no
U	no	no	no

Benefits of Update Locks

T1	T2	T3	T4	T5	T6	T7	T8	T9
s(A);r(A)								
	s(A);r(A)							
		s(A);r(A)						
			s(A);r(A)					
				ul(A);r(A)				
					s(A);denied			
						s(A);denied		
							s(A);denied	
								s(A);denied
u(A)								
	u(A)							
		u(A)						
			u(A)					
				xl(A);w(A)				
				u(A)				
					s(A);r(A)			
						s(A);r(A)		
							s(A);r(A)	
								s(A);r(A)

T5 will wait for T1-T4 to release the lock on A, but will stop T6-T9 obtaining a lock on A as otherwise its updating of the lock to exclusive would be blocked for a long time (until all the other transactions release the lock on A).

Recovery from Crashes

UNDO

Action	t	Buff A	Buff B	A in HD	B in HD	Log
Read(A,t)	8	8		8	8	<Start T>
t:=t*2	16	8		8	8	
Write(A,t)	16	16		8	8	<T,A,8>
Read(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
Write(B,t)	16	16	16	8	8	<T,B,8>
Flush Log						
Output(A)	16	16	16	16	8	
Output(B)	16	16	16	16	16	<Commit T>
Flush Log						

U1: <T,X,o> must be on disk **before** output(X).
U2: <COMMIT T> must be on disk **after** all output(X) by T.

REDO:

Action	t	Buff A	Buff B	A in HD	B in HD	Log
Read(A,t)	8	8		8	8	<Start T>
t:=t*2	16	8		8	8	
Write(A,t)	16	16		8	8	<T,A,16>
Read(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
Write(B,t)	16	16	16	8	8	<T,B,16>
						<Commit T>
Flush Log						
Output(A)	16	16	16	16	8	
Output(B)	16	16	16	16	16	

R1: All <T,…,…> (including <COMMIT T>) must be on disk **before** any output(X) by T.

UNDO/REDO:

Action	t	Buff A	Buff B	A in HD	B in HD	Log
Read(A,t)	8	8		8	8	<Start T>
t:=t*2	16	8		8	8	
Write(A,t)	16	16		8	8	<T,A,8,16>
Read(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
Write(B,t)	16	16	16	8	8	<T,B,8,16>
Flush Log						
Output(A)	16	16	16	16	8	<Commit T>
Output(B)	16	16	16	16	16	
Flush Log						

UR1: <T,X,o,n> must be on disk **before** output(X) by T.

Undo vs Redo vs Undo/Redo

U1: <T,X,o> must be on disk **before** output(X).

U2: <COMMIT T> must be on disk **after** all output(X) by T.

R1: All <T,…,…> (including <COMMIT T>) must be on disk **before** any output(X) by T.

UR1: <T,X,o,n> must be on disk **before** output(X) by T.

If there is a crash

- UNDO: **undo incomplete transactions**
  - Complete transactions have their changes to disk already. No need to worry about them.

- REDO: **redo complete transactions**
  - Incomplete transactions don’t have any change to disk. They can be safely ignored.

- UNDO/REDO: **undo incomplete transactions, redo complete transactions**.

Undo with NQ Checkpointing

<START T1>

<T1,A,5>

<START T2>

<T2,B,10>

<START CKPT (T1,T2)>

<T2,C,15>

<START T3>

<T1,D,20>

<COMMIT T1>

<T3,E,25>

<COMMIT T2>

<END CKPT>

<T3,F,30>

Wait until active transactions are committed before writing <END CKPT>

What if we have a crash right after **<T3,E,25>**?
Care about T3 and T2.

← A crash occurs at this point. Care only about T3.

Redo with Checkpointing

<START T1>
<T1,A,5>
<START T2>
<COMMIT T1>
<T2,B,10>
<START CKPT(T2)>
<T2,C,15>
<START T3>
<T3,D,20>
<END CKPT>
<COMMIT T2>
<COMMIT T3>

The buffer containing A might be **dirty.** If so, copy it to disk. Then write **<END CKPT>.**

During this period three other actions took place.

Undo/Redo with Checkpointing

```
<START T1>
<T1,A,4,5>
<START T2>
<COMMIT T1>
<T2,B,9,10>
<START CKPT(T2)>
<T2,C,14,15>
<START T3>
<T3,D,19,20>
<END CKPT>
<T2,E,20,21>
<COMMIT T2>
<COMMIT T3>   ← A crash occurs just before this
```

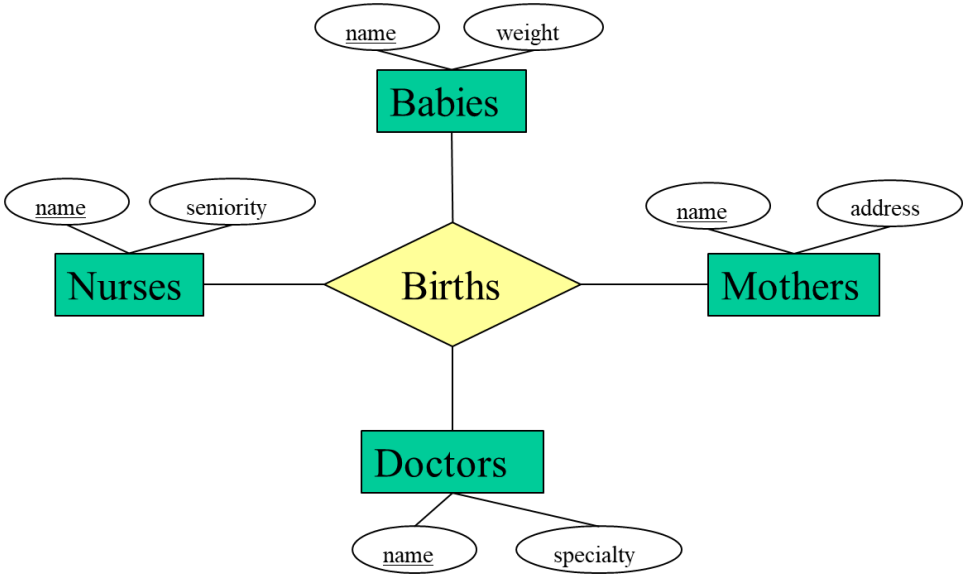
- Suppose the crash occurs just before <COMMIT T3>.
- We identify T2 as committed but T3 as incomplete.
- It is *not* necessary to set B to 10 since we know that this change reached disk before **<END CKPT>.**
- However, we need to REDO E; set E=21.
- Also, we need to UNDO T3; set D=19

Pros and Cons

- Undo requires the data to be written to disk in order to commit a transaction
  - this increases the # of disk I/O's
- Redo requires keeping all modified blocks buffered until after transaction commits
  - this increases the average # of buffers needed by transactions
- Undo/Redo requires bigger log entries.

Database Normalization (BCNF)

Births Table



Births(baby, mother, nurse, doctor)

Some facts and assumptions

a) For every baby, there is a unique mother.

b) For every (existing) combination of a baby and a mother there is a unique doctor.

c) There are many nurses assisting in a birth.

Anomalies

Baby	Mother	Nurse	Doctor
Ben	Mary	Ann	Brown
Ben	Mary	Alice	Brown
Ben	Mary	Paula	Brown
Jason	Mary	Angela	Smith
Jason	Mary	Peggy	Smith
Jason	Mary	Rita	Smith

- **Redundancy**.
  - Information may be repeated unnecessarily in several tuples.

Functional Dependencies

- $X \rightarrow A$ for a relation R means that
  - whenever two tuples of R agree on all the attributes of X, then they must also agree on attribute A.
  - We say: "X functionally determines A"

Example

baby → mother

baby mother → doctor

Convention:

X, Y, Z represent sets of attributes;
A, B, C,… represent single attributes.
will write just ABC, rather than $\{A,B,C\}$.

Keys of Relations

- K is a superkey for relation R if K functionally determines **all** of R's attributes.

- K is a key for R if
  - K is a superkey, and
  - Can't remove some attribute from K and still be superkey

- E.g. K=**{baby, nurse}** is a **key** for **Births**.

Boyce-Codd Normal Form

- Boyce-Codd Normal Form (BCNF): simple condition under which the anomalies can be guaranteed not to exist.

- A relation R is in **BCNF** if:

  Whenever there is a nontrivial dependency

  $A_1 \ldots A_n \rightarrow B_1 \ldots B_m$

  for R, it must be the case that

  {$A_1, \ldots, A_n$**}** is a **superkey** for R.

Babies isn't in **BCNF**.

- FD: baby→mother
- Left side isn't a superkey.
  - We know: baby doesn't functionally determine nurse.

Decomposition into BCNF

- One is **all the attributes** involved in the **violating dependency**
- The other is the **left side** and **all the other attributes** not involved in the dependency.

- By **repeatedly**, choosing suitable decompositions, we can break any relation schema into a collection of smaller schemas in BCNF.

Babies Example

Births(baby, mother, nurse, doctor)

baby→mother is a violating FD, so we decompose.

Baby	Mother
Ben	Mary
Jason	Mary

Baby	Nurse	Doctor
Ben	Ann	Brown
Ben	Alice	Brown
Ben	Paula	Brown
Jason	Angela	Smith
Jason	Peggy	Smith
Jason	Rita	Smith

This relation needs to be further decomposed using the baby→doctor FD.
We, will see a formal algorithm for deducing this FD.

Rules About Functional Dependencies

- Suppose **we are told** of a set of functional dependencies that a relation satisfies.

  Based on them we can **deduce** other dependencies.

Example.

baby→mother and

baby mother → doctor

imply

baby → doctor

But, what's the algorithm?

Algorithm: Starting with a set of attributes, repeatedly expand the set by adding the right sides of **FD**'s as soon as we have included their left sides.

If $B \in \{A_1, A_2, \dots, A_n\}^+$ then FD: $A_1A_2\dots A_n \rightarrow B$ **holds**.

If $B \notin \{A_1, A_2, \dots, A_n\}^+$ then FD: $A_1A_2\dots A_n \rightarrow B$ **doesn't hold.**

$\{A_1, A_2, \dots, A_n\}$ is a **superkey** iff $\{A_1, A_2, \dots, A_n\}^+$ is the set of **all** attributes.

Movie Example

Movies(title, year, studioName, president, presAddr)

and FDs:

title year → studioName

studioName → president

president → presAddr

Last two violate **BCNF**. Why?

Compute {title, year}+, {studioName}+, {president}+ and see if you get all the attributes of the relation.

If not, you got a BCNF violation, and need to decompose.

Example (Continued)

Let's decompose starting with:

studioName → president

Optional rule of thumb:

Add to the right-hand side any other attributes in the closure of studioName.

{studioName}+ = {studioName, president, presAddr}

Thus, we get:

studioName→president presAddr

Example (Continued)

Using: studioName→president presAddr we decompose into:

Movies1(studioName, president, presAddr)

Movies2(title, year, studioName)

Movie2 is in **BCNF**.

What about Movie1?

FD president→presAddr violates **BCNF**.

Why is it bad to leave Movies1 as is?

*If many studios share the same president than we would have redundancy when repeating the **presAddr** for all those studios.*

Example (Continued)

We decompose Movies1, using FD: president→presAddr

The resulting relation schemas, both in **BCNF**, are:

Movies11(president, presAddr)

Movies12(studioName, president)

So, finally we got Movies11, Movies12, and Movies2.

Must keep applying the decomposition rule as many times as needed, until all our relations are in **BCNF**.