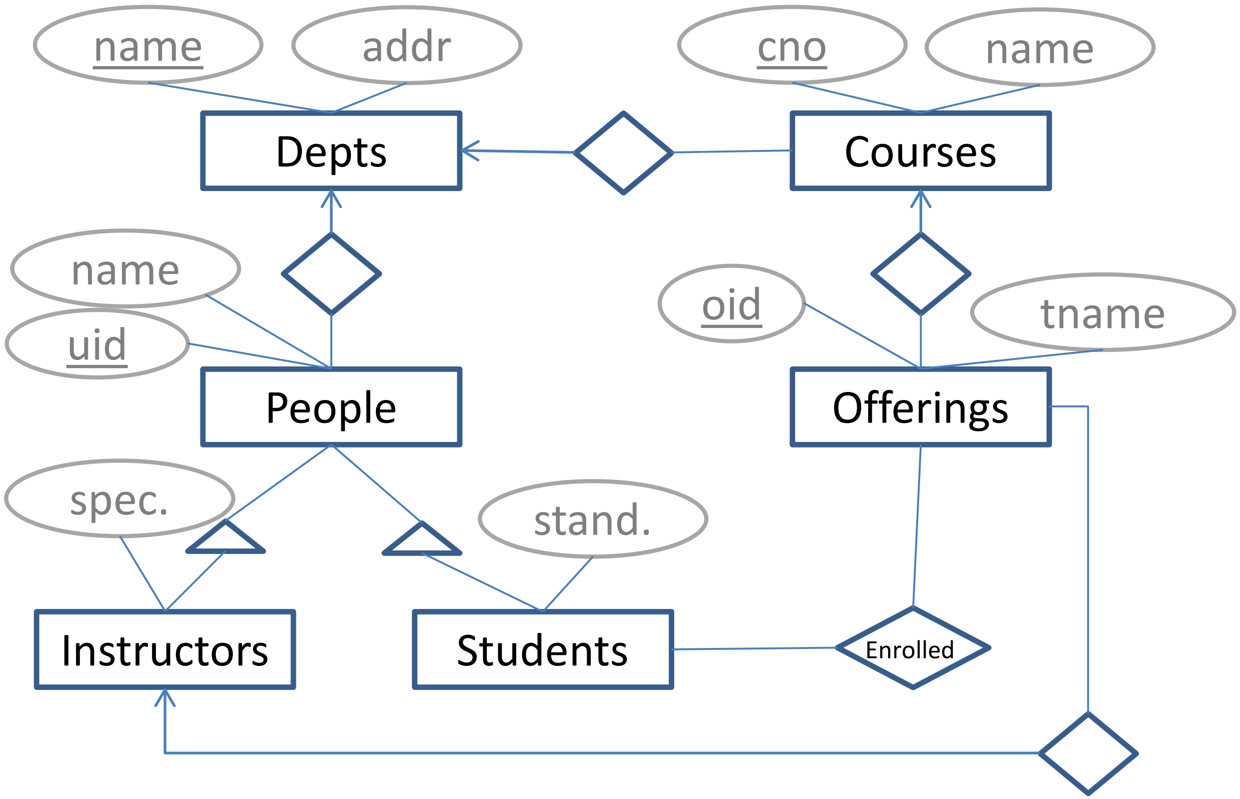
Review

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');
```

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)
     ON DELETE CASCADE
);
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)
     ON DELETE CASCADE
);
```

Views with check option: Example

```
CREATE TABLE Hotel (
  room_nbr INT NOT NULL,
  arrival_date DATE NOT NULL,
  departure_date DATE NOT NULL,
  guest_name CHAR(15) NOT NULL,
  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 OUTER JOIN

RIGHT OUTER 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

- Using **Movies**, **StarsIn**, and **Stars**,
  find the star's total length of film played.
  We are interested only in Canadian stars and
  who first appeared in a movie before 2000.

```
SELECT starName, SUM(length)
FROM Movies, StarsIn, Stars
WHERE Movies.title=StarsIn.title AND Movies.year=StarsIn.year
      AND Stars.name=StarsIn.starName
      AND Stars.birthplace LIKE '%Canada%'
GROUP BY starName
HAVING MIN(StarsIn.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;
```

Intersection, Union, Difference

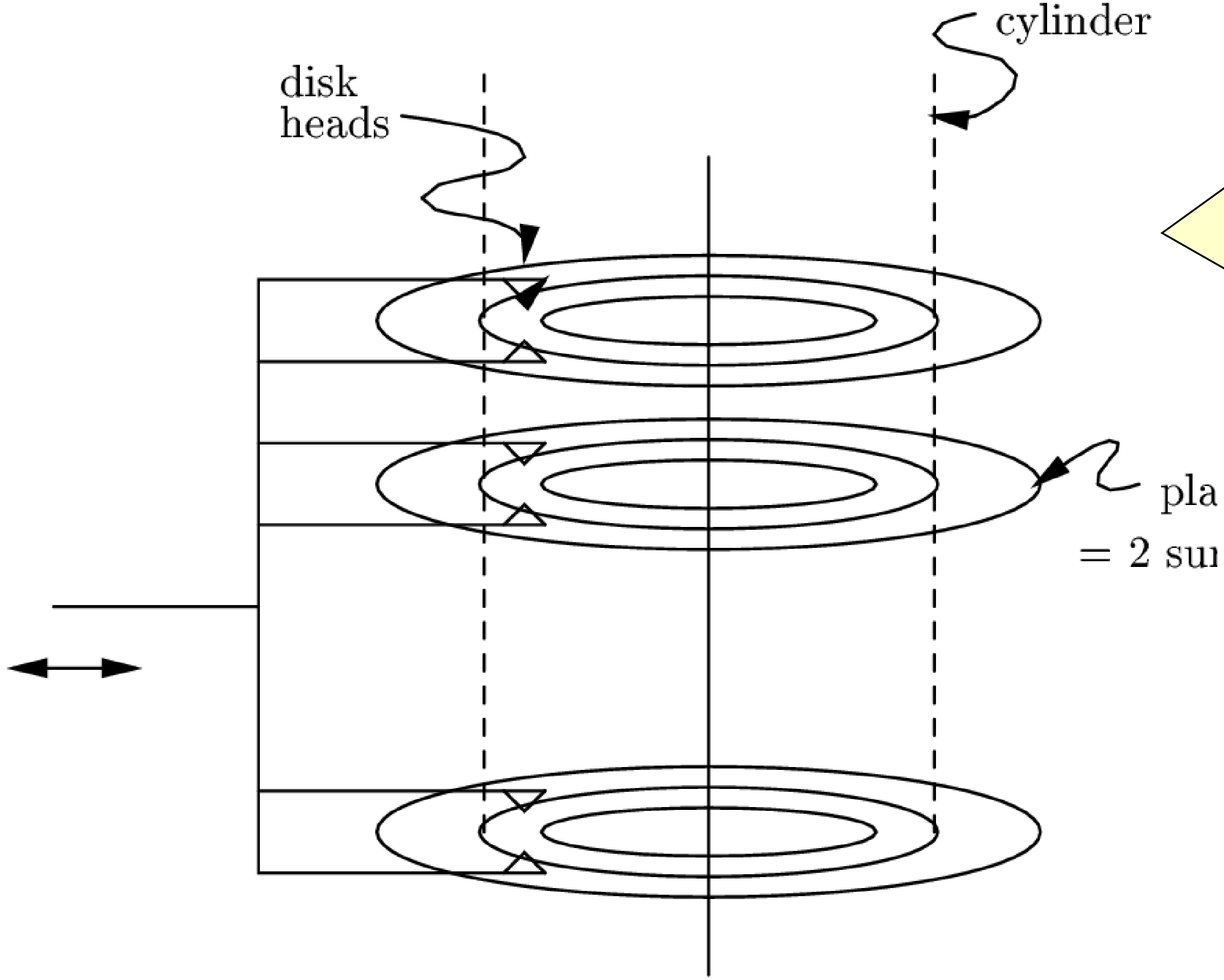
```
SELECT title, year
FROM StarsIn
WHERE starName='Richard Gere'
  INTERSECT
SELECT title, year
FROM StarsIn
WHERE starName='Julia Roberts' ;
```

```
SELECT title, year
FROM StarsIn
WHERE starName='Richard Gere'
  UNION
SELECT title, year
FROM StarsIn
WHERE starName='Julia Roberts' ;
```

```
SELECT title, year
FROM StarsIn
WHERE starName='Richard Gere'
  EXCEPT
SELECT title, year
FROM StarsIn
WHERE starName='Julia Roberts' ;
```

Storage

Disks



- Platters with top and bottom surfaces rotate around a spindle.
- 2--30 surfaces.
- Rotation speed: 3600--7200 rpm.
- One head per surface.
- All heads move in and out in unison.

AVG time to read a 16,384-byte block (Megatron 747 – a fictitious, but realistic disk…)

- Transfer time is **0.13** milliseconds and
- Average rotational latency 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

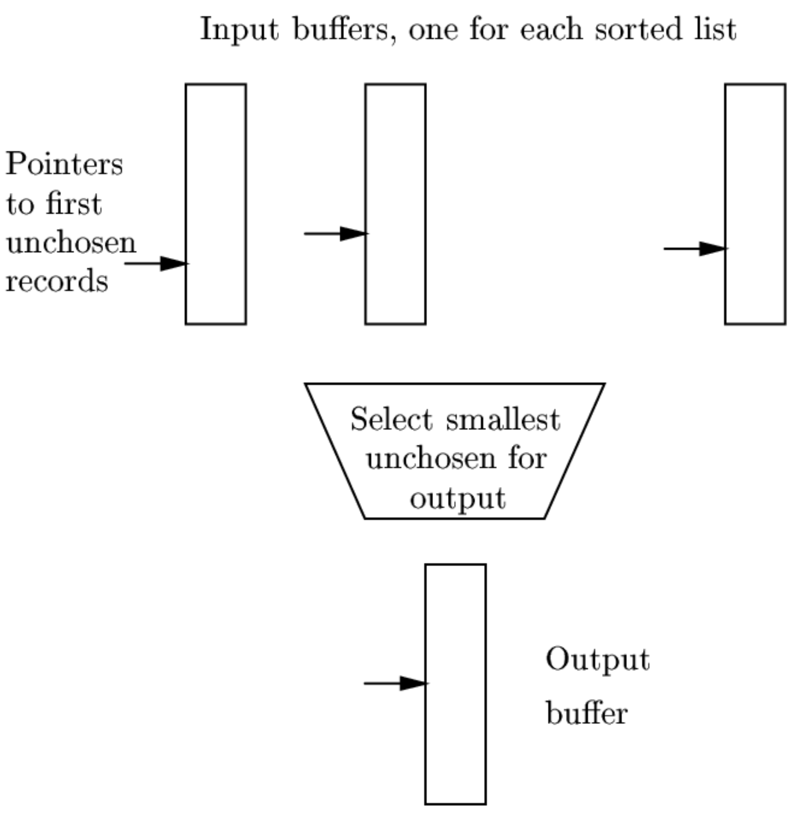
- Total: 6.46 + 4.17 + 0.13 = 10.76 ms (or about 11 ms)

2PMMS

Phase 1

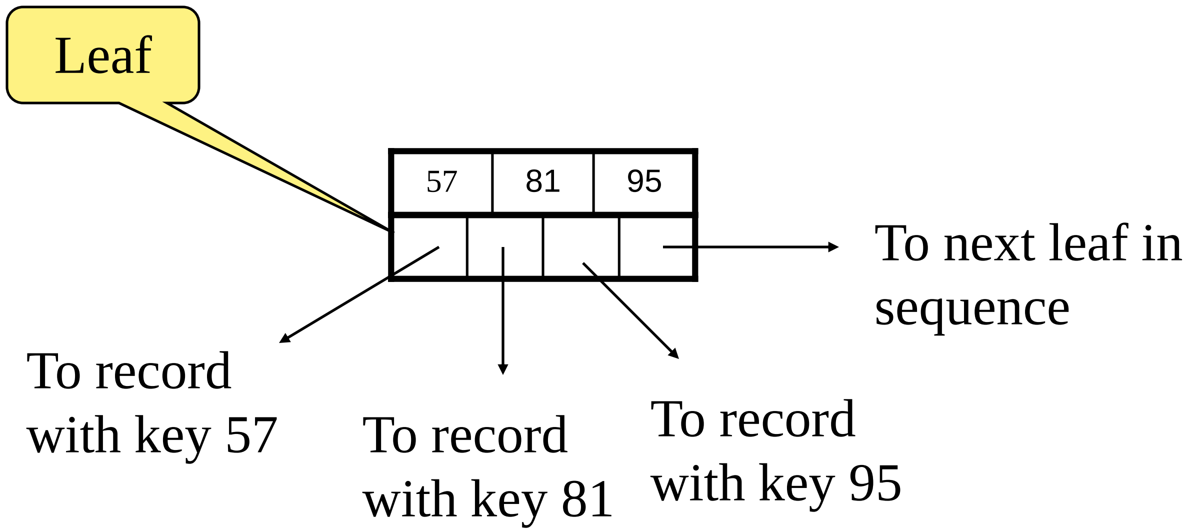
1. Fill main memory with records.
2. Sort using favorite main memory sort.
3. Write sorted sublist to disk.
4. Repeat until all records have been put into one of the sorted lists.

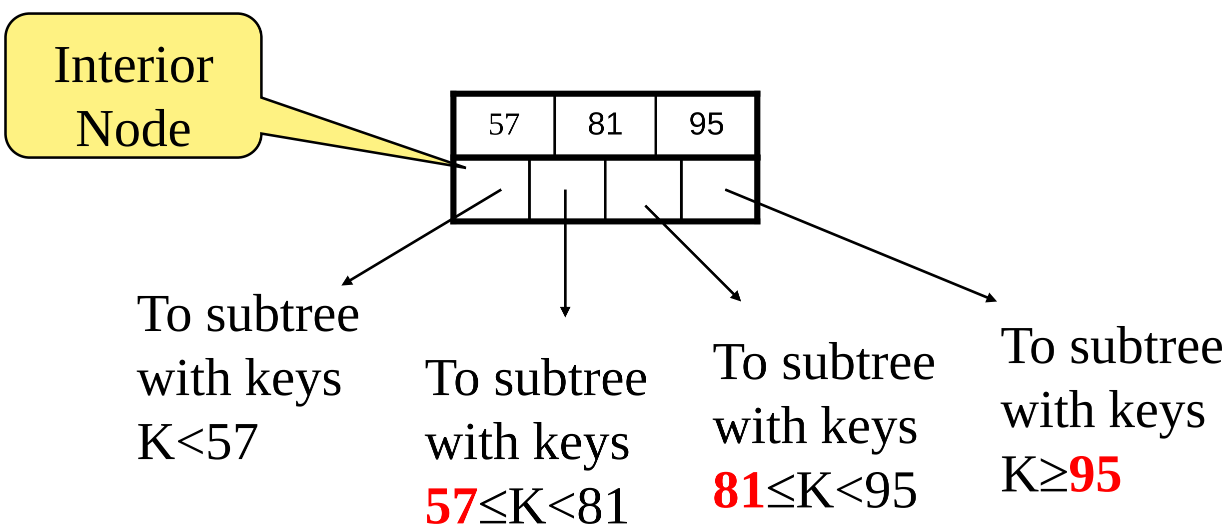
Phase 2



Indexes

BTrees: A typical leaf and interior node (unclustered index)

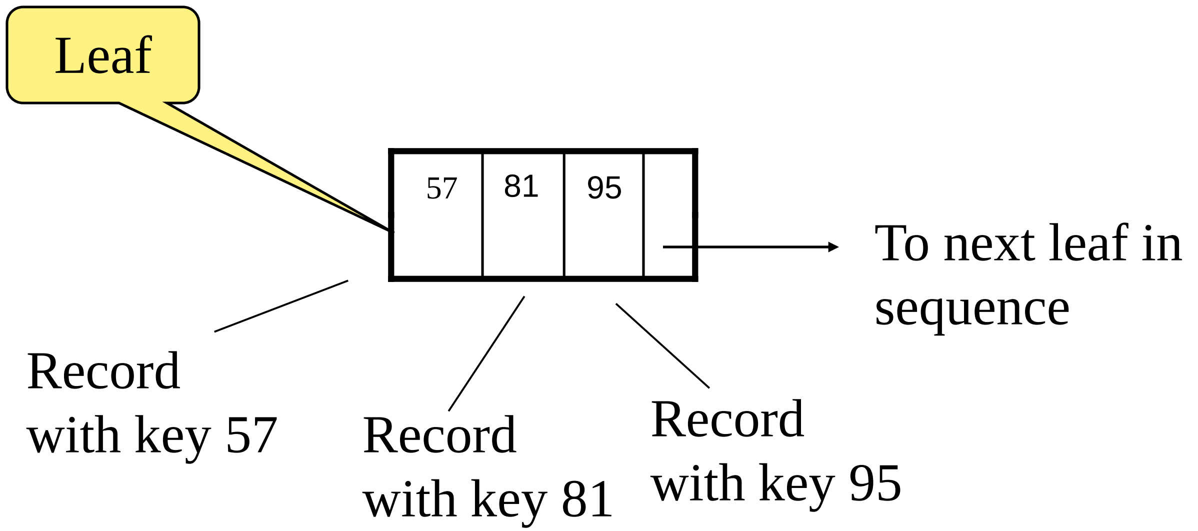


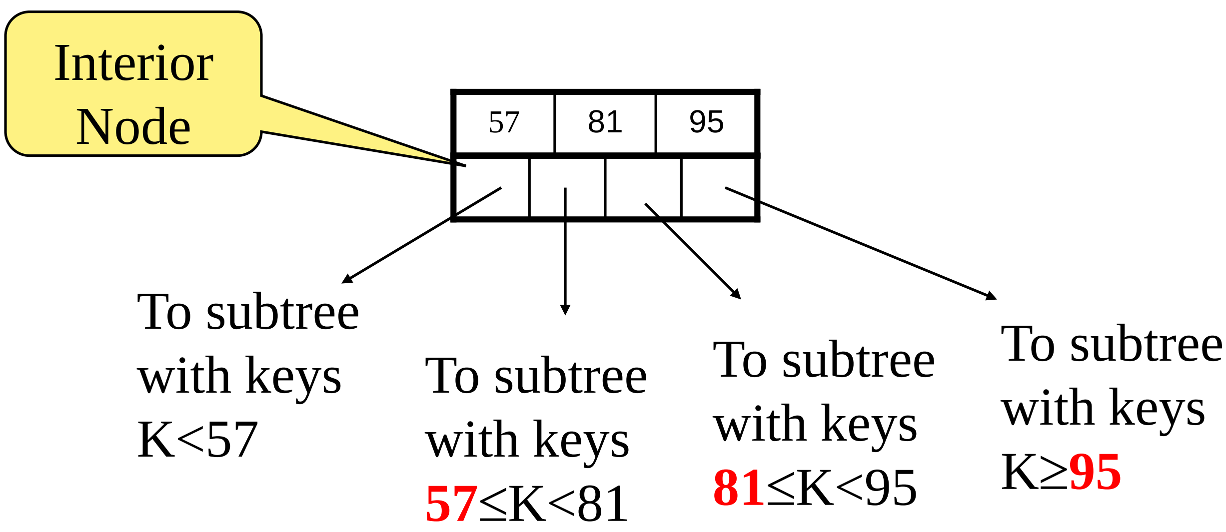


57, **81**, and **95** are **the least keys** we can reach by via the corresponding pointers.

A typical leaf and interior node (clustered index)

A clustered index is by necessity primary.

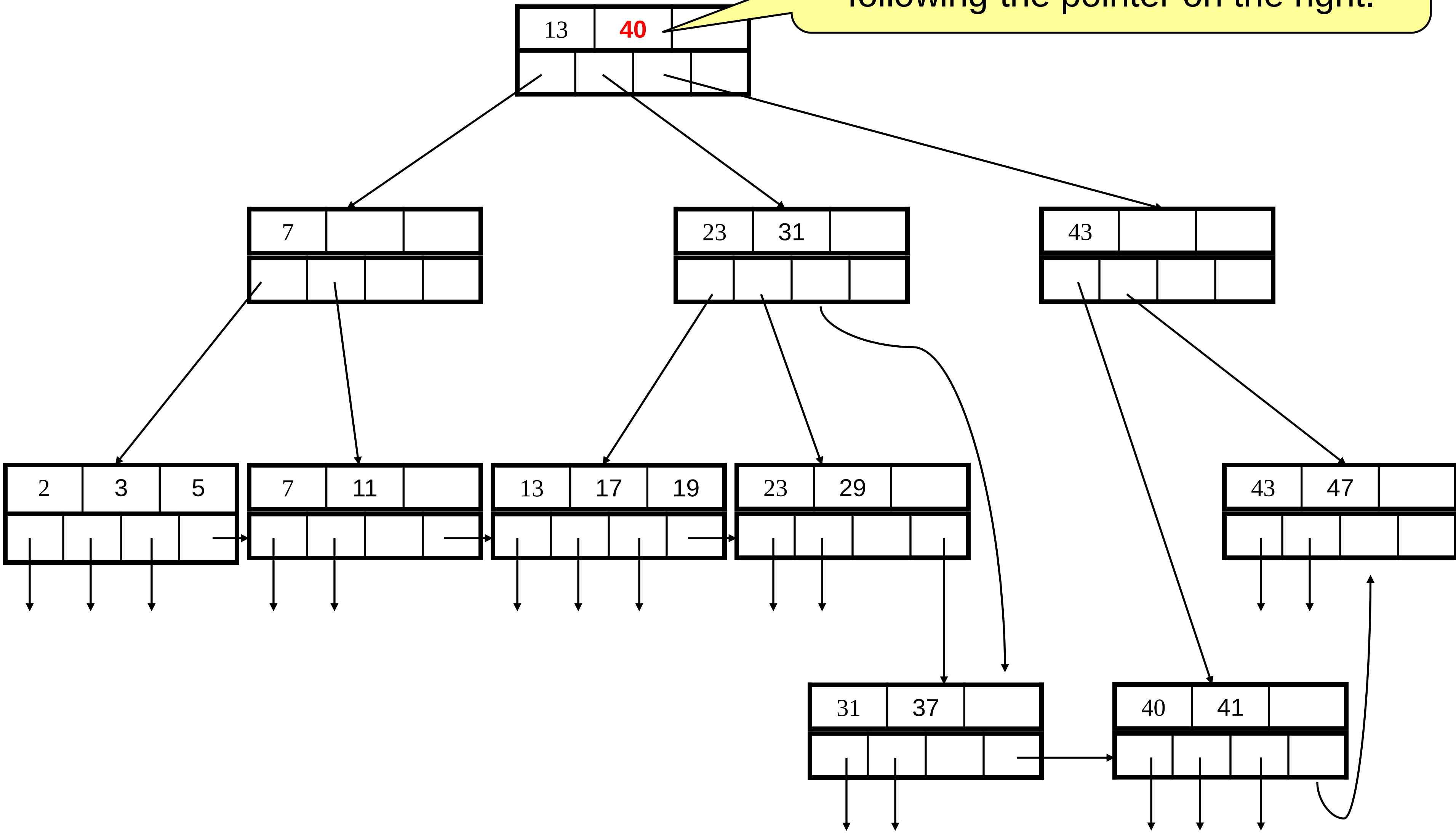




57, **81**, and **95** are **the least keys** we can reach by via the corresponding pointers.

Insertion of key 40

40 is the least key reachable by following the pointer on the right.



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:

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

$$\Rightarrow \; n \leq 409$$

Query Evaluation

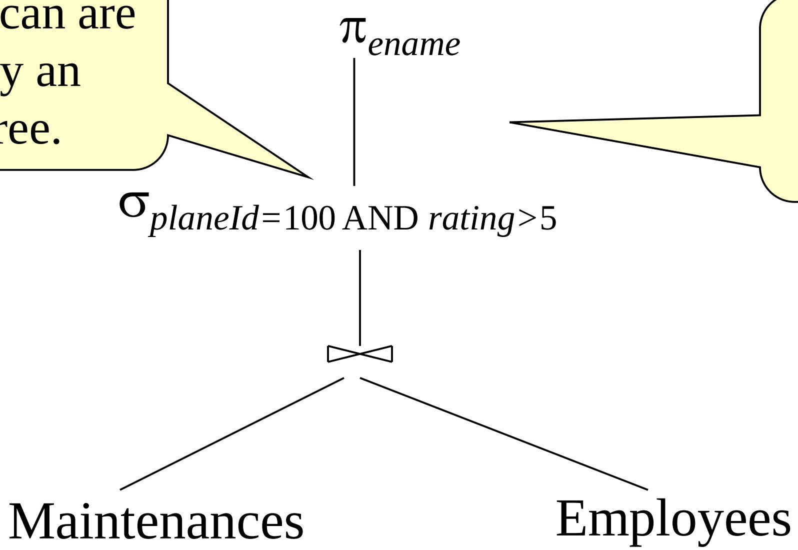
An SQL query and its RA equiv.

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;

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

RA expressions can are represented by an expression tree.



An algorithm is chosen for each node in the expression tree.

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

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

Running Example – Airline

Employees (<u>sin INT</u>, ename VARCHAR(20), rating INT, address VARCHAR(90))

Maintenances (<u>sin INT</u>, <u>planeId INT</u>, <u>day DATE</u>, 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.

Index nested loops join

- Scan **Maintenances** and for each tuple **probe** **Employees** for matching tuples (using the index on **Employees.SIN**).

- Analysis:
  - For each of the 100,000 maintenance tuples, we try to access the corresponding employee with 3 I/Os (via the index).
  - So, 100,000*3 = 300,000 I/Os !!

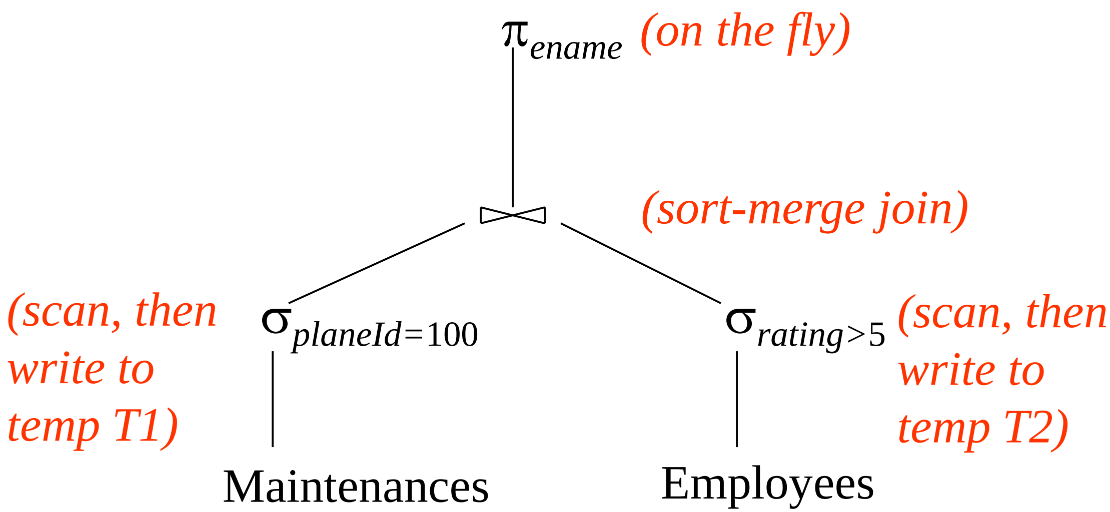
Sort-merge join

- **Sort** both tables **on the join column**, then scan them to find matches

- Analysis:
  - Sort Maintenances with 2PMMS, and Employees with 2PMMS
  - Cost for sort is
    - 4 * 1000 = 4000 I/Os for Maintenances and
    - 4 * 50 = 200 I/Os. for Employees
  - Then we merge-join. This requires an additional scan of both tables.
  - Thus the total cost is 4000+ 200+ 1000+ 50= **5250** I/Os. (Much better!!)

- However, "**index nested loops**" method has the nice property that it is **incremental**.

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

Transactions

Controlling Concurrent Behavior

Summarizing the Terminology

- **Transaction** (model) is a *sequence* of *r* and *w* actions on database elements.
- **Schedule** is a *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**: A schedule whose "**effect**" is equivalent to that of some serial schedule.
- **Conflict-serializable Schedule** if it can be converted into a serializable schedule with the **same effect** by a series of non-conflicting swaps of adjacent elements

We talk about a **sufficient condition** for serializability.

Example of a conflict-serializable schedule

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

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

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

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

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

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

The operations in bold can be safely swapped.

Precedence graphs

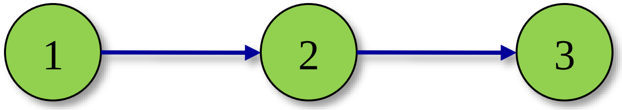
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

Note the following:

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

➢These are conflicts since they contain a read/write on the same element

➢They cannot be swapped. Therefore $T_1 < T_2 < T_3$



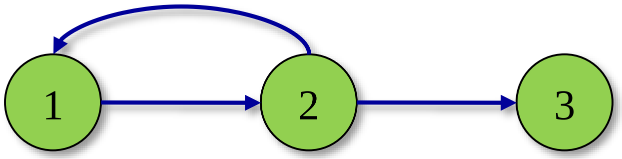
Conflict serializable

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

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

Legal Schedule Doesn't Mean Serializable

A **scheduler** takes requests from transactions for reads and writes, and decides if it is "OK" to allow them to operate on DB or defer them until it is safe to do so.

One approach is to use locks.

T_1	T_2	A	B
		25	25
$l_1(A)$; $r_1(A)$			
A = A + 100			
$w_1(A);u_1(A)$		125	
	$l_2(A);r_2(A)$		
	A = A * 2		
	$w_2(A);u_2(A)$	250	
	$l_2(B);r_2(B)$		
	B = B * 2		
	$w_2(B);u_2(B)$		50
$l_1(B);r_1(B)$			
B = B + 100			
$w_1(B);u_1(B)$			150

Consistency constraint assumed for this example: A=B

Two Phase Locking

There is a 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 (if there are no shared locks anymore).
  - 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 Logging

log all “important actions”

<START T> -- transaction **T** started.

<T,X,Old$_X$> -- database element **X** was modified;
it used to have the value Old$_X$

<COMMIT T> -- transaction **T** has completed

<ABORT T> -- transaction **T** couldn’t complete successfully.

Two rules:

U1: Log records for element X must be on disk **before** any change to X appears on disk.

U2: If a transaction T commits, then <**COMMIT T**> must be written to log in disk **after** all elements changed by T are written to disk.

Force the log to be saved to disk by executing Flush Log.

Example:

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						

Recovery With Undo Logging

Examine each log entry <T, X, v>

a) If T complete, do nothing.

b) If T is incomplete, restore the old value of X

In what order?

From most recent to earliest.

Nonquiescent Checkpoint (NQ CKPT)

- **Solution:** Let $(T_1,...,T_k)$ be the active transactions

  1. Write <START CKPT($T_1,...,T_k$)>
     and flush log to disk.

  2. Wait until all $T_1…T_k$ commit or abort,
     but don't prohibit new transactions.

  3. When all $T_1…T_k$ are "done", write <END CKPT> and flush log to disk.

Recovery with NQ CKPT

First case:

If the crash follows **<END CKPT>**

Then, undo

any incomplete transaction that

started after **<START CKPT>**

Second case:

If the crash occurs between **<START CKPT>** and **<END CKPT>**

Then, undo

any incomplete transaction that

*is on the **CKPT** list or*

started after **<START CKPT>**

Undo Drawback

- Can’t commit a transaction without first writing all its changed data to disk.

- Sometimes we can save disk I/O if we let changes to the DB reside only in main memory for a while;
- …as long as we can fix things up in the event of a crash…

Redo Logging

log all “important actions”

<START T> -- transaction **T** started.

<T,X,New$_X$> -- database element **X** was modified;
the new value is **New$_X$**

<COMMIT T> -- transaction **T** has completed

<ABORT T> -- Transaction **T** couldn’t complete successfully.

One rule:

R1. Before outputting X to disk, all log entries (including <COMMIT T>) must be written to log in disk.

Example:

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	

Recovery With Redo Logging

Only committed transactions matter!

Examine each log entry <T, X, v>

a) If T incomplete, do nothing.

b) If T is complete, redo the operation:

For each **<T, X, v>** in the log do:

WRITE(X,v); OUTPUT(X);

In what order?

From the earliest to latest.

Checkpointing for Redo

1. Write a <START CKPT(T_1,...,T_k)> record to the log, where T_i's are all the active transactions.

2. Write to disk all the *dirty buffers* of transactions that had already committed when the START CKPT was written to log.

3. Write an <END CKPT> record to log.

Recovery with Ckpt. Redo

Two cases:

1. If the crash follows **<END CKPT>,**

   we can **restrict** ourselves to transactions that began after **<START CKPT>** and those in the **START** list.

   This is because we know that, in this case, every value written by committed transactions, before <**START CKPT(…)**>**,** is now in disk.

2. If the crash occurs between **<START CKPT>** and **<END CKPT>,**

   then go and find the previous **<END CKPT>** and do the same as in the first case.

Functional Dependencies

Superkeys and Keys

- *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,
  but no proper subset of *K* is a superkey.

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 $B \in \{A_1, A_2, \ldots, A_n\}^+$ then FD: $A_1A_2\ldots A_n \to B$ **holds.**

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

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

Movie Example

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

and FDs:

title year $\rightarrow$ studioName

studioName $\rightarrow$ president

president $\rightarrow$ 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.

In general, we must keep applying the decomposition rule as many times as needed, until all our relations are in **BCNF**.

Data Analysis with SQL Window Functions

Table omc

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
…	…	…	…	…

For each category: which months were revenues below the average of the current year?

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

Detail

Window
In this example: all the tuples with the same **year** and **cat** as in the detail part.

Second: Extract what you want with enclosing query

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

Which months did the revenues from a product category drop below those of the same month of the previous year?

```
SELECT *
FROM (
  SELECT year, month, cat, revenue,
    LAG(revenue,12) OVER (PARTITION BY cat
                          ORDER BY year, month)
                                AS prev_year_rev
  FROM omc )
WHERE revenue < prev_year_rev
ORDER BY 1,2;
```

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

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

Similar to ROW_NUMBER, but ties are not broken.
See next slides for results.

If there were no ties,
ROW_NUMBER, RANK and
DENSE_RANK would be the same.