Final Review

Database APIs

Database libraries: Provide an API to a DB

Cursors: Iteratively page through results

"Impedance mismatches"

- Types: different ints, floats, dates, etc
- Object: Objects/structs != rows
- Constraints: program logic VS DB rules

SQL Injection: Building strings instead of letting API do it

Duplication is bad

One type of meat, cheese, and vegetable

Pizza	Topping	Type
1	Mozzarella	Cheese
1	Pepperoni	Meat
1	Olives	Vegetable
2	Mozzarella	Cheese
2	Sausage	Meat
2	Peppers	Vegetable

Key? (Pizza, Type)

Anomalies

Update: Correct a type, must update all rows

Insert: Add a type without a pizza?

Delete: Cancel a pizza; lose a type!

Decomposition

Replace schema R with 2+ smaller schemas that

1. each contain subset of attrs in R
2. together include all attrs in R

ABCD replaced with AB, BCD or AB, BC, CD

Desirable properties

1. Lossless join: able to recover R from smaller relations
2. Dependency preserving: enforce constraints on by only enforcing constraints on smaller schemas (no joins)

Functional Dependencies (FD)

$X \rightarrow Y$

holds on R

if $t_1.X = t_2.X$ then $t_1.Y = t_2.Y$

where X,Y are subsets of attrs in R

Examples of FDs in person-hobbies table

sid → name, address

hobby → cost

sid, hobby → name, address cost

Keys and Functional Dependencies

A key must determine all column values

Must be the left hand side (aka input or source) of FDs

$BC \rightarrow N$, $N \rightarrow BO$

NBO key is N: determines all other fields

- Banker name determines branch, office
- NBO is in BCNF: All FDs are keys

Where do FDs come from?

thinking really hard aka application semantics

can’t stare at database to derive (like ICs)

Like a Mathematics conjecture:

one counter example can disprove, but examples can’t prove there are no example in the universe

Closure of FDs

If I know

Name → Bday and Bday → age

Then it implies

Name → age

An FD f’ is implied by set F if f’ is true when F is true

F^+: the closure of F is all FDs implied by F

Can we construct this closure automatically? YES

Closure of FDs

Inference rules called Armstrong's Axioms

- Reflexivity: if $Y \subseteq X$ then $X \to Y$
- Augmentation: if $X \to Y$ then $XZ \to YZ$ for any Z
- Transitivity: if $X \to Y$ & $Y \to Z$ then $X \to Z$

These are sound and complete rules

- sound: doesn't produce FDs not in the closure
- complete: doesn't miss any FDs in the closure

Closure of FDs

F = {A→B, B→C, CB→E}

Is A→E in the closure?

A → B	given
A → AB	augmentation A
A → BB	apply A→B
A → BC	apply B→C
BC → E	given
A → E	transitivity

Minimum Cover of FDs

1. Turn FDs into *standard form*

   Split FDs so there is one attribute on the right side

2. Minimize left side of each FD

   For each FD, check if can delete a left attribute using another FD

   given ABC → D, B→C   can reduce to AB→D, B→C

3. Delete redundant FDs

   check each remaining FD and see if it can be deleted

   e.g., in closure of the other FDs

2 must happen before 3!

Normal Forms

Criteria met by a relation R wrt functional dependencies

Boyce Codd Normal Form (BCNF)

No redundancy, may lose dependencies

Third Normal Form (3NF)

May have redundancy, no decomposition problems

Lossless Join Decomposition

join the decomposed tables to get *exactly the* original

e.g., decompose R into tables X, Y

$\pi_X(R) \bowtie \pi_Y(R) = R$

Lossless wrt F if and only if F^+ contains

$X \cap Y \rightarrow X$ or $Y \cap X \rightarrow Y$

intersection of X, Y is a key for one of them

Lossless Join Decomposition

Lossless wrt F if and only if F^+ contains

$X \cap Y \rightarrow X$ or $Y \cap X \rightarrow Y$

intersection of X,Y is a key for one of them

FDs: $A \rightarrow C, A \rightarrow B$

A	B	C
1	2	1
5	3	4
9	2	6

→

A	B
1	2
5	3
9	2

B	C
2	1
3	4
2	6

→

A	B	C
1	2	1
5	3	4
9	2	6
1	2	6
9	2	1

Lossy! $AB \cap BC = B$ doesn't determine anything

Dependency-preserving Decomposition

F_R = Projection of F onto R

FDs $X \rightarrow Y$ in F^+ s.t. X and Y attrs are in R

Subset of F that are "valid" for R

If R decompose to X, Y.

FDs that hold on X, Y equivalent to all FDs on R

$(F_X \cup F_Y)^+ = F^+$

Consider ABCD, C is key, $AB \rightarrow C$, $D \rightarrow A$

BCNF decomposition: BCD, DA

$AB \rightarrow C$ doesn't apply to either table!

BCNF

Relation R in BCNF has *no redundancy* wrt FDs

(only FDs are key constraints)

```
F: set of functional dependencies over relation R
X: Subset of attributes of R
A: One attribute of R
   for (X→A) in F
      A is in X (trivial/reflexivity) OR
      X is a superkey of R
      (superkeys include candidate keys)
```

BCNF

```
while BCNF is violated
    R with FDs F_R
    if X→Y violates BCNF
        turn R into R-Y & XY
```

3NF

F^{min} = minimal cover of F

Run BCNF using F^{min}

for $X \rightarrow Y$ in F^{min} not in projection onto $R_1 \ldots R_N$

  create relation XY

BCNO $BC \rightarrow N$, $N \rightarrow BO$

  NBO, CN using $N \rightarrow BO$

Disks

Time to access (read or write) a disk block

seek time	2-4 msec avg (arm movement)
rotational delay	2-4 msec (based on rotation speed)
transfer time	0.3 msec/64kb page

Throughput

read	~150 MB/sec
write	~50 MB/sec

Key: reduce seek and rotational delays

HW & SW approaches

What is a page?

Unit of transfer between storage and database

Typically fixed size

Small enough for one I/O to be fast

Big enough to not be wasteful

Usually a multiple of 4 kB

- Intel virtual memory hardware page size
- Modern disk sector size (minimum I/O size)

Records and Files

Record: "application" storage unit

- e.g. a row in a table

Page: Collection of records

File: Collection of pages

- insert/delete/modify record
- get(record_id) a record
- scan all records

May be in multiple OS files spanning multiple disks

Unordered Heap Files

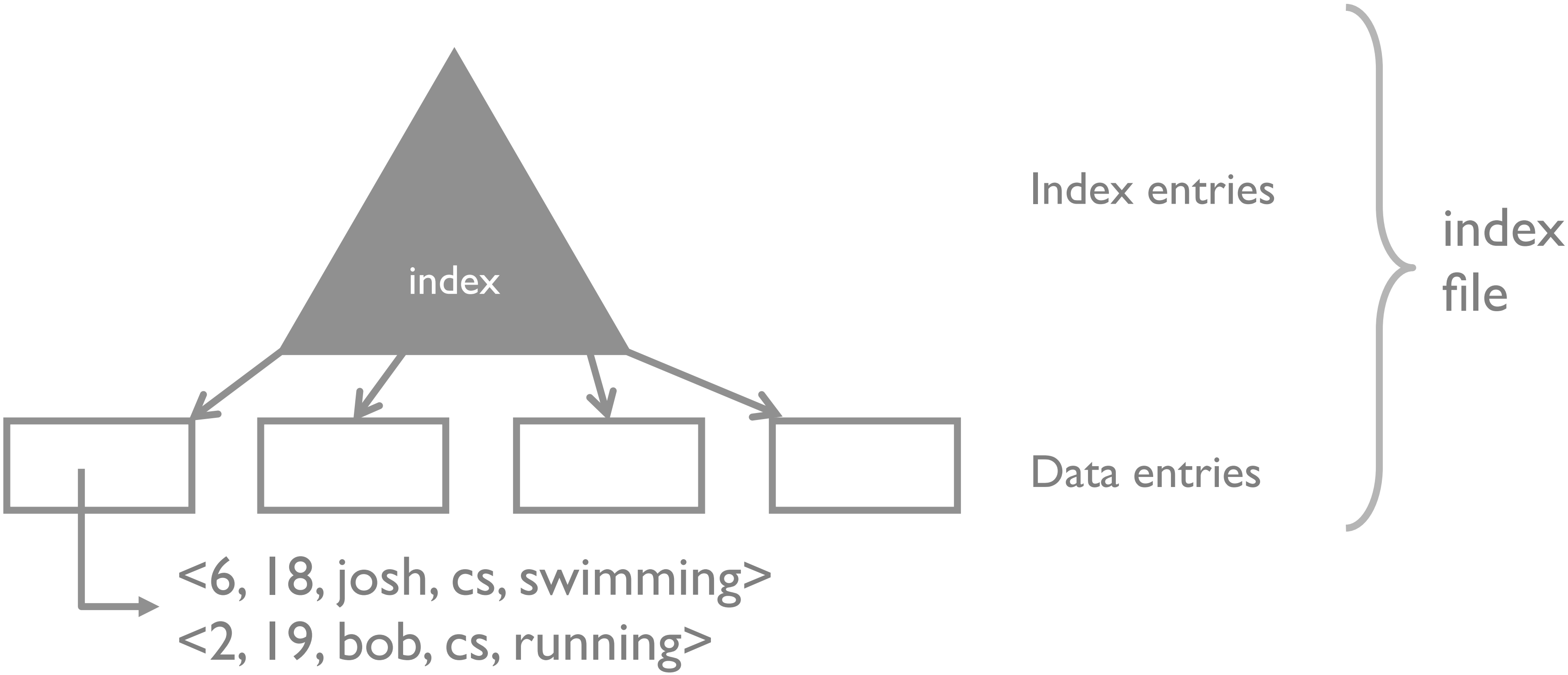
Unordered collection of records

Pages allocated as collection grows

Need to track:

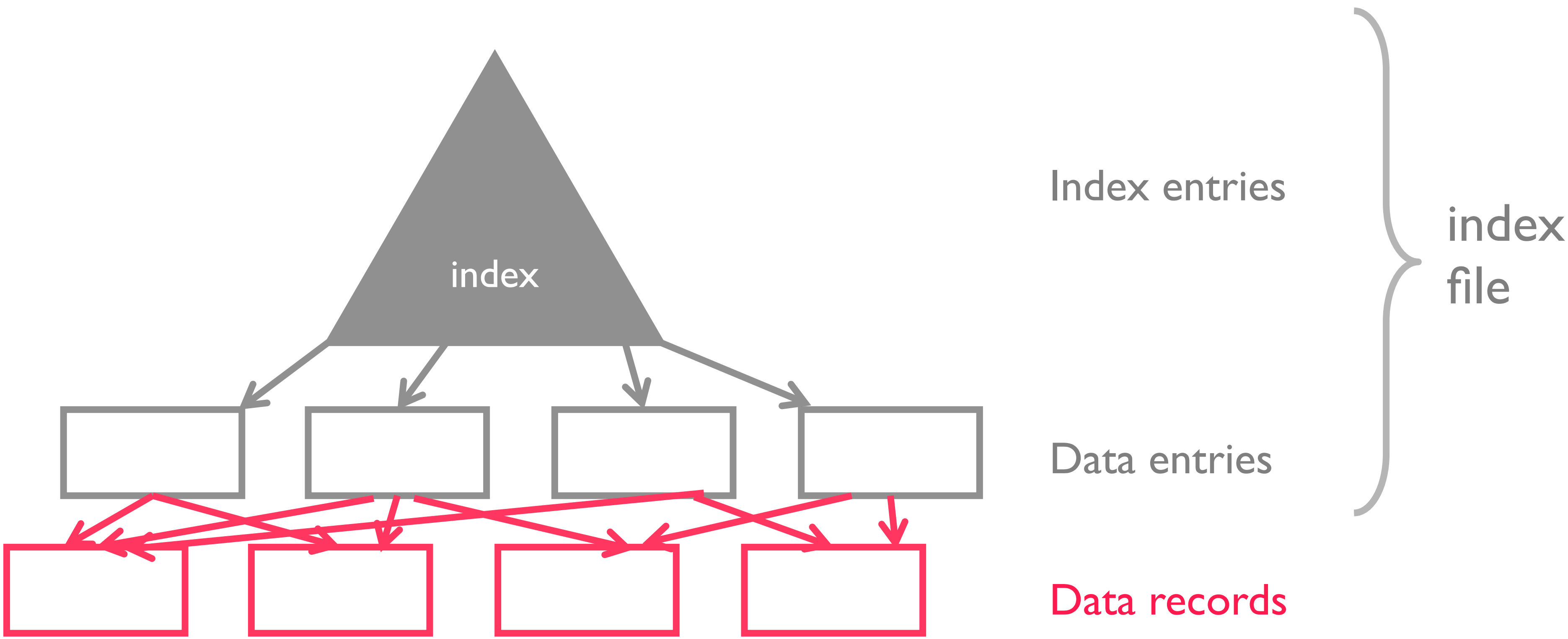
- pages in file
- free space on pages
- records on page

High level (Primary) index structure



What is a data entry?
actual data record

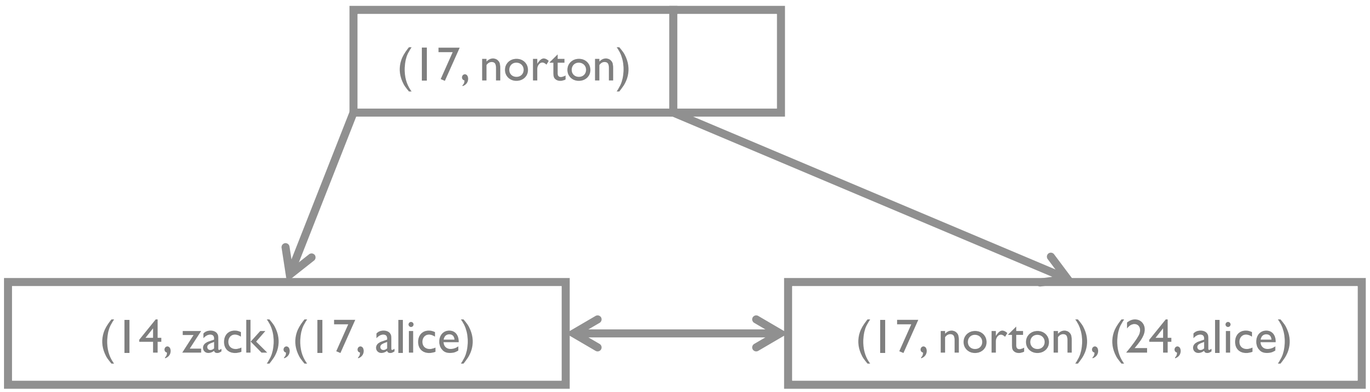
High level (Secondary) index structure



What is a data entry?
  actual data record
  <search key value, rid>

Tradeoffs
directly access tuple.
compact, fixed size entries

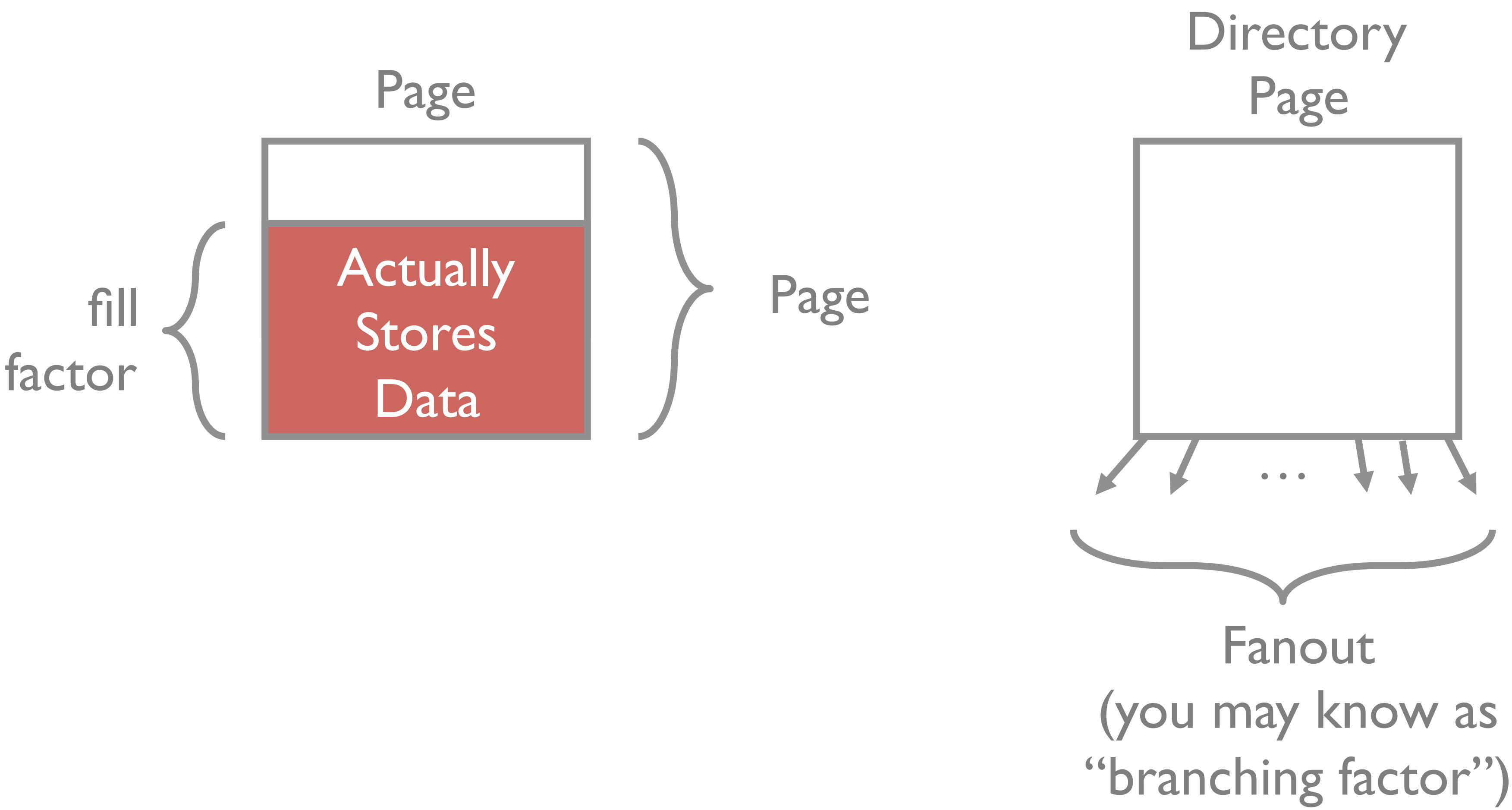
B+ Tree on (age, name)



How do the following queries use the index on (age, name)?

```
SELECT age   WHERE age = 14
SELECT *     WHERE age < 18 AND name < 'monica'
SELECT age   WHERE name = 'bobby'
```

Terminology



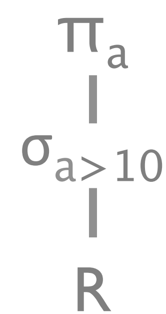
SQL → Query Plan

SELECT a FROM R

$\pi_a(R)$

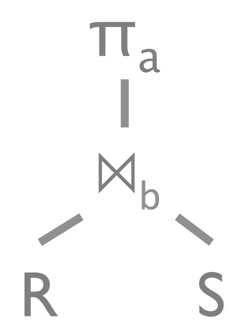
SELECT a FROM R
WHERE a > 10

$\pi_a(\sigma_{a>10}(R))$



SELECT a
FROM R JOIN S
ON R.b = S.b

$\pi_a(\bowtie_b(R,S))$



Query Evaluation

Naïve execution (operator at a time)

read R

filter a>10 and write out

read and project a

Cost: B + M + M

```
SELECT a
FROM R
WHERE a > 10
```

$$\pi_a$$
$$|$$
$$\sigma_{a>10}$$
$$|$$
$$R$$

B # *data* pages

M # pages matched in WHERE clause

Could we do better?

Query Evaluation

Pipelined exec (tuple/page at a time)

- read first page of R, pass to σ
- filter a > 10 and pass to π
- project a
- (all operators run concurrently)
- Cost: B

Note: can't pipeline some operators!

e.g., sort, some joins, aggregates

why?

SELECT a
FROM R
WHERE a > 10

$$\begin{array}{c} \pi_a \\ | \\ \sigma_{a>10} \\ | \\ R \end{array}$$

B # *data* pages

M # pages matched in WHERE clause

Predicate Push Down

SELECT a
FROM R
WHERE a > 10

$$\begin{array}{c}\pi_a \\ | \\ \sigma_{a>10} \\ | \\ R \\ \\ \text{(a)}\end{array} \qquad \begin{array}{c}\sigma_{a>10} \\ | \\ \pi_a \\ | \\ R \\ \\ \text{(b)}\end{array}$$

Which is faster if B+ Tree index: (a) or (b)?

(a) $\log_F(B)$ + M pages

(b) B pages

It's a Good Idea, especially when we look at Joins

Nested Loops Join:

```
# outer ⋈₁ inner
# outer JOIN inner ON outer.1 = inner.1
for row in outer:
    for irow in inner:
        if row[0] == irow[0]:   # could be any check
            yield (row, irow)
```

Very flexible

- Equality check can be replaced with any condition
- Incremental algorithm
- Cost: M + MN

Is this the same as a cross product?

Nested Loops Join

What this means in terms of disk IO

tableA join tableB; tableA is "outer"; tableB is "inner"
M pages in tableA, N pages in tableB, T tuples per page

$M + T \times M \times N$

for each tuple *t* in tableA, (M pages, TM tuples)
    scan through each page *pi* in the inner (N pages)
        compare all the tuples in *pi* with *t*

Indexed Nested Loops Join

```
for row in outer:
   for irow in index.get(row[0], []):
      yield (row, irow)
```

Slightly less flexible

- Only supports conditions that the index supports

Sort Merge Join

Sort outer and inner tables on join key

- Cost: 2-3 scans of each table

Merge the tables and compute the join

- Cost: 1 scan of each table

Overall Properties

- cost: 3(M+N) to 4(M+N)
- results are sorted
- highly sequential access
- (weapon of choice for very large datasets)

Cost Estimation

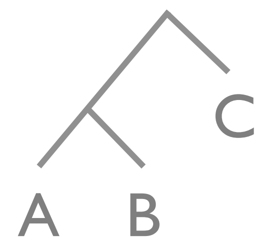
estimate(operator, inputs, stats) → cost

estimate cost for each operator

- depends on input *cardinalities* (# tuples)
- discussed earlier in lecture

estimate output size for each operator

- need to call estimate() on inputs!
- use selectivity. assume attributes are independent



Try it in PostgreSQL: EXPLAIN <query>;

Estimate Size of Output

```
Emp:   1000 Cardinality
Dept:  10 Cardinality

Cost(Emp join Dept)
```

Naïve

# total records	1000 * 10	= 10,000
Selectivity of Emp	1 / 1000	= 0.001
Selectivity of Dept	1 / 10	= 0.1
Join Selectivity	1 / max(1k, 10)	= 0.001
Output Card:	10,000 * 0.001	= 10

note: selectivity defined wrt cross product size

Note: estimate wrong if this is a key/fk join on emp.did = dept.did: 1000 results

Transactions

Sequence of actions treated as a single unit

Atomicity: Apply all changes or none

("atomic" because it is indivisible)

Solves the crash problem

Isolation: Illusion that each transaction executes sequentially, without concurrency

Transaction Guarantees

Atomicity
"all or nothing": All changes applied, or none are
users never see in-between transaction state

Consistency
database always satisfies Integrity Constraints
Transactions move from valid database to valid database

Isolation:
from transaction's point of view, it's the only one running

Durability:
if transaction commits, its effects *must persist*

Concepts

Serial schedule

One transaction at a time. no concurrency.

Equivalent schedule

the database state is the same at end of both schedules

Serializable schedule (gold standard)

equivalent to a serial schedule

Why Serializable Schedule? Anomalies

Reading in-between (uncommitted) data

T1:	R(A) W(A)		R(B) W(B) abort
T2:		R(A) W(A) commit	

WR conflict or dirty reads

Reading same data gets different values

T1:	R(A)		R(A) W(A) commit
T2:		R(A) W(A) commit	

RW conflict or unrepeatable reads

Why Serializable Schedule? Anomalies

Stepping on someone else's writes

- T1: W(A) W(B) commit
- T2: W(A) W(B) commit
- WW conflict or lost writes

Notice: all anomalies involve writing to data that is read/written to.

- If we track our writes, maybe can prevent anomalies

Conflict Serializability

def: a schedule that is conflict equivalent to a serial schedule

Meaning: you can swap non-conflicting operations to derive a serial schedule.

$\forall$ conflicting operations O1 of T1, O2 of T2

- O1 always before O2 in the schedule or
- O2 always before O1 in the schedule

Conflict Serializability

Transaction Precedence Graph

Edge Ti → Tj if:

1. Ti read/write A before Tj writes A or
2. Ti writes some A before Tj reads/writes A

If graph is acyclic (does not contain cycles) then conflict serializable!

Fine, but what about COMMITing?

T1	R(A)	W(A)			R(B)	ABORT
T2			R(A)	COMMIT		

Not recoverable

Promised T2 everything is OK. IT WAS A LIE.

T1	R(A)	W(B)	W(A)			ABORT
T2				R(A)	W(A)	

Cascading Rollback.

T2 read uncommitted data → T1's abort undos T1's ops & T2's

Two-Phase Locking (2PL)

Growing phase: acquire locks

Shrinking phase: release locks

Guarantees serializable schedules!

shrink here

T1	R(A) W(B) W(A)		ABORT
T2		R(A) W(A)	

Uh Oh, same problem

Lock-based Concurrency Control

Strict two-phase locking (Strict 2PL)

- Growing phase: acquire locks
- Shrinking phase: release locks
- Release locks after commit/abort

Why? Which problem does it prevent?

T1	R(A) W(B)	W(A)			ABORT
T2			R(A) W(A)		

Avoids cascading rollbacks!

Deadlocks

T1	R(A) W(A)			W(B)?
T2		R(B) W(B)		W(A)?

Possible for a cycle of transactions to wait forever

Typical solution: abort txn if waiting too long
(lock timeout)

Concept: Sequence of changes

Before any change: write it to a sequential file
(write ahead of the change: write-ahead log)

On commit/abort: write "commit/abort" record

On recovery: replay complete transactions

Requirements: ordered; no "holes"

Redo-only log

Rule: To write a modified page to disk:

- All writers must have committed
  (no uncommitted changes that might be aborted)
- All log records on disk

On abort: Need to undo changes

Keep "undo" information in memory

When do disk writes happen?

On commit: Write log to disk, wait to complete (sequential IO)

After commit: Write modified pages when needed, or when idle

“free” the log? Complicated; not covered

Disadvantage: Big transactions

What about transactions bigger than memory?

Need to write uncommitted changes to disk

Solution: ARIES algorithm (IBM again; 1992)

Idea: Write both undo and redo records

Can write page if undo records on disk

Redo and Undo log

Rule: To write a modified page to disk:

- Log undo and redo records on disk

On commit: Write all undo/redo records for txn

Write pages at any time after log records on disk!

Aries Recovery Algorithm

3 phases

- Analyze the log to find status of all xacts
  - Committed or in flight?
- Redo xacts that were committed
  - Now at the same state at the point of the crash
- Undo partial (in flight) xacts

Recovery is *extremely* tricky and *must be correct*