Final Review

CS348 Spring 2025:
Introduction to Database Management

Instructor: **Xiao Hu**

Sections: 001, 002, 003

Final Exam

- Logistics
  - Time & Date: 4:00 PM – 6:30 PM August 5
  - Location: PAC GYM
  - All contents in Lectures 1 – 20 (no optional parts)
  - Closed book, but a four-page reference sheet will be provided (already released on Learn)

- How to prepare (in addition to lectures)?
  - A1, A2, A3
  - Midterm questions and partial solutions
  - More exercise questions (but NOT samples)

Final Exam

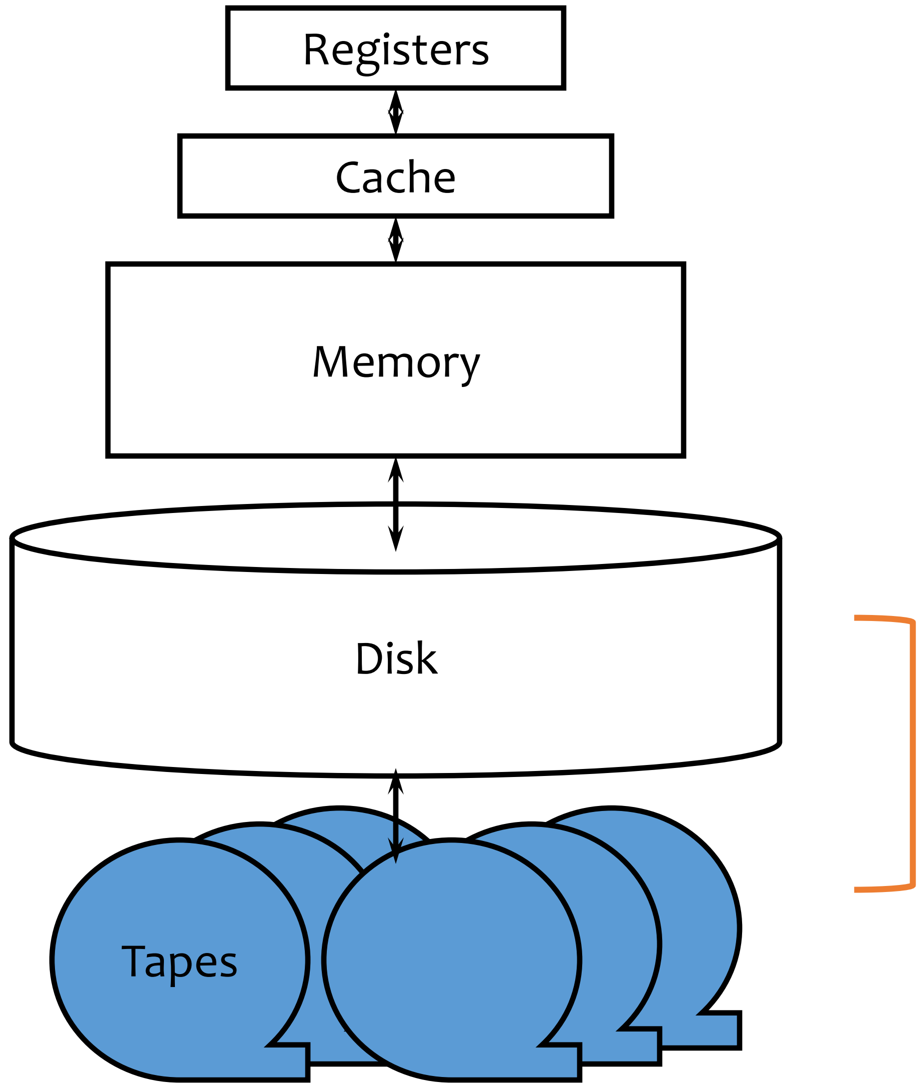
- Total: 110 points (10 bonus points)
- Midterm topics (around 45 points)
  - Relational model and relational algebra
  - SQL
  - Database design

See midterm review

- Database internals (around 55 + 10 points)
  - Physical data design
  - Indexing
  - Query processing & optimization
  - Transaction

Physical Data Storage

Storage hierarchy



Secondary storage

Tertiary storage

Non-volatile

Disk access time

Disk access time (= Seek time + Rotational delay): the duration from when a read or write request is issued until data transfer begins

- Seek time: time for disk heads to move to the correct track
- Rotational delay: time for the desired block to appear under the disk head
- Transfer time: time to read/write data in the block (= time for the disk to rotate over the block)

Data access time = Seek time + Rotational delay + Transfer time

Random v.s. Sequential disk access

Random disk access: successive requests are for blocks that are randomly located on disk

- Average seek time (~ 5 ms)
- Average rotational delay (~ 4.2 ms)

Sequential disk access: successive requests are for successive blocks that are on the same track or adjacent tracks

- Seek time and rotational delay are 1 time delay
- Easily an order of magnitude faster than random disk access!

Record layout: fixed-length fields

- All field lengths and offsets are constant
  - Computed from schema, stored in the system catalog

```
CREATE TABLE User(uid INT, name CHAR(20), age INT, pop FLOAT);
```

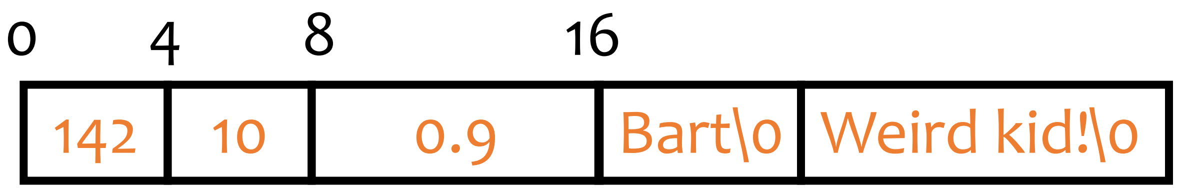
  - If block size != 36, one record may be split across multiple blocks, or moved to the next block (by leaving the remaining space empty)
- What about NULL?
  - Add a bitmap at the beginning of the record

Record layout: variable-length records

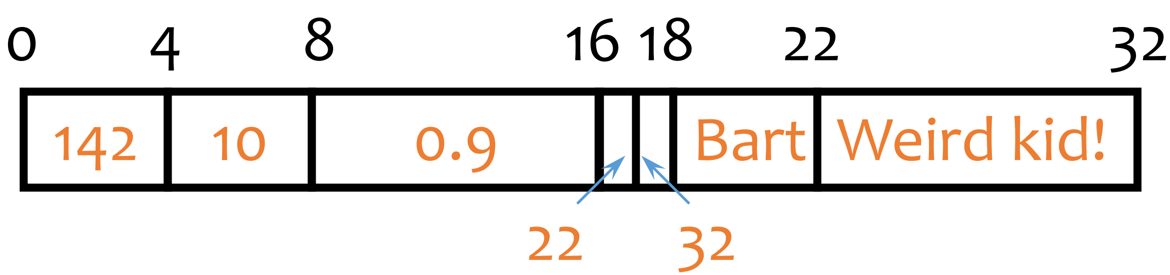
Put all variable-length fields at the end

```
CREATE TABLE User(uid INT, name VARCHAR(20), age INT,
    pop FLOAT, comment VARCHAR(100));
```

- Approach 1: use field delimiters ('\0' okay?)



- Approach 2: use an offset array



- Scheme update is messy if length of a field changes

BLOB fields

```
CREATE TABLE User(uid INT, name VARCHAR(20), age INT,
            pop FLOAT, comment VARCHAR(100),
BLOB(32000));
```

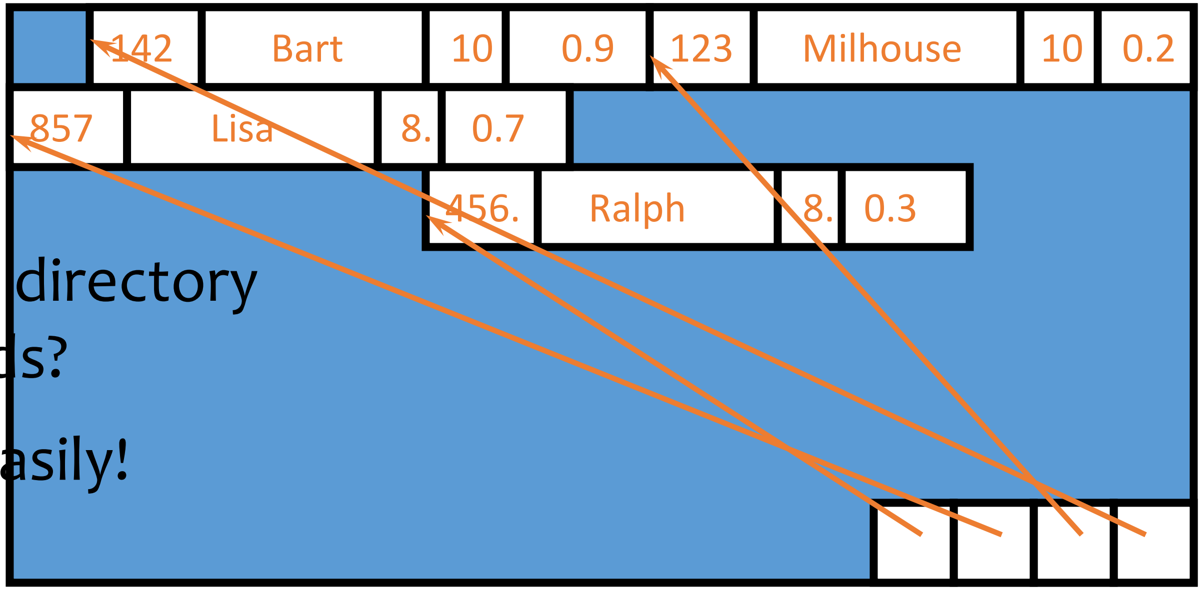
- User records get "de-clustered"
  - Bad because most queries do not involve picture

- Decomposition (automatically and internally done by DBMS without affecting the user)
  - (*<u>uid</u>*, *name*, *age*, *pop*)
  - (*<u>uid</u>*, *picture*)

Similar to Translating ISA:
Entity-in-all-superclasses

Block layout - (N-ary Storage Model)

- Store records from the beginning of each block
- Use a directory at the end of each block
  - To locate records and manage free space
  - Necessary for variable-length records



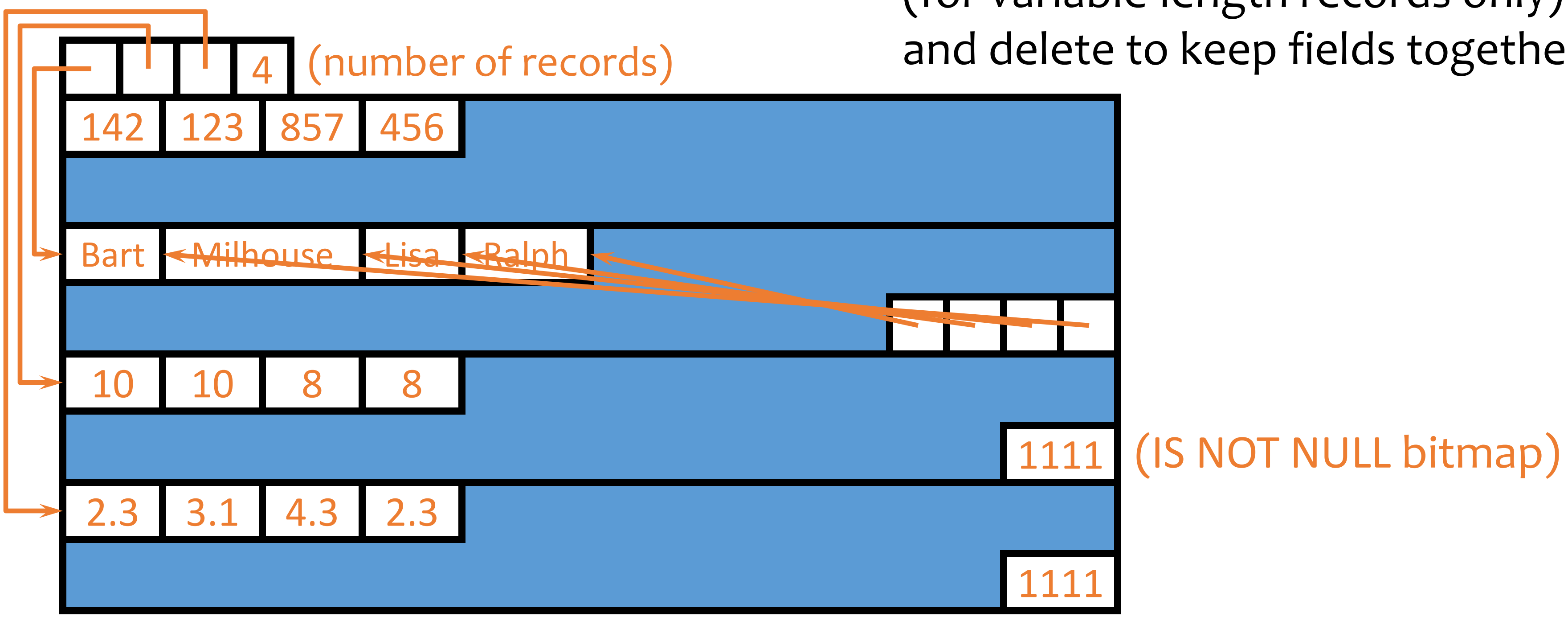
Why store data and directory at two different ends?

So both can grow easily!

Block layout - Partition Attributes Across

- Most queries only access a few columns
- Cluster values of the same columns in each block
- Better sequential reads for queries that read a single column

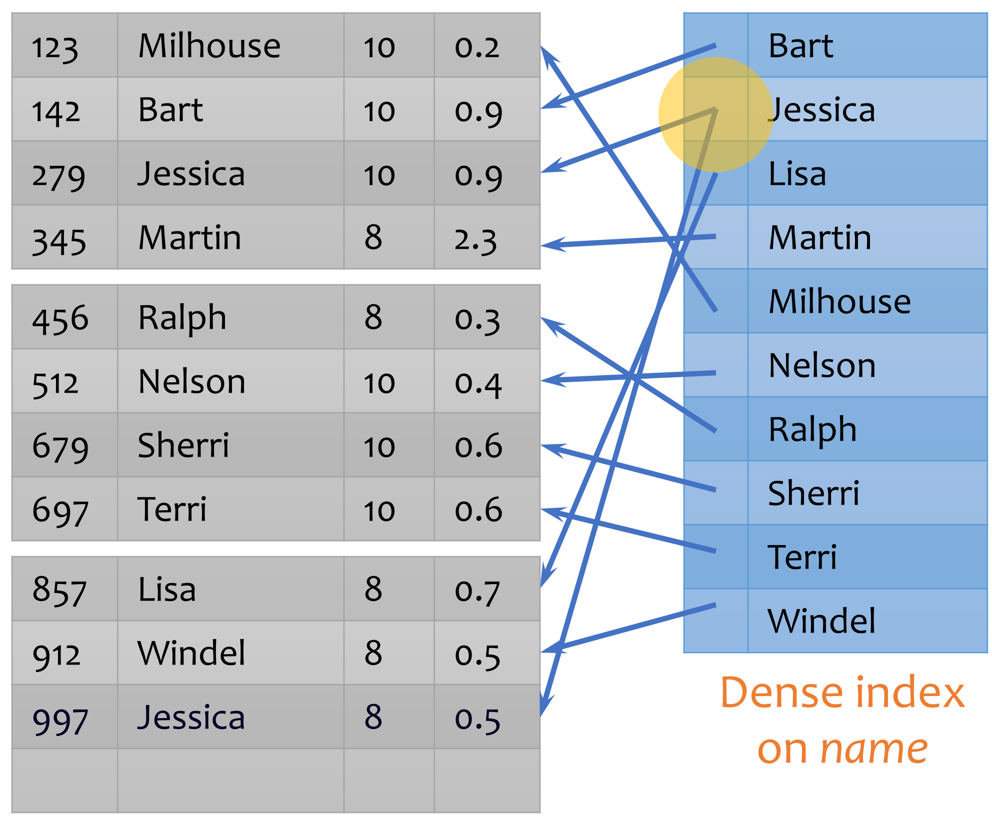
Reorganize after every update (for variable-length records only) and delete to keep fields together



Indexes

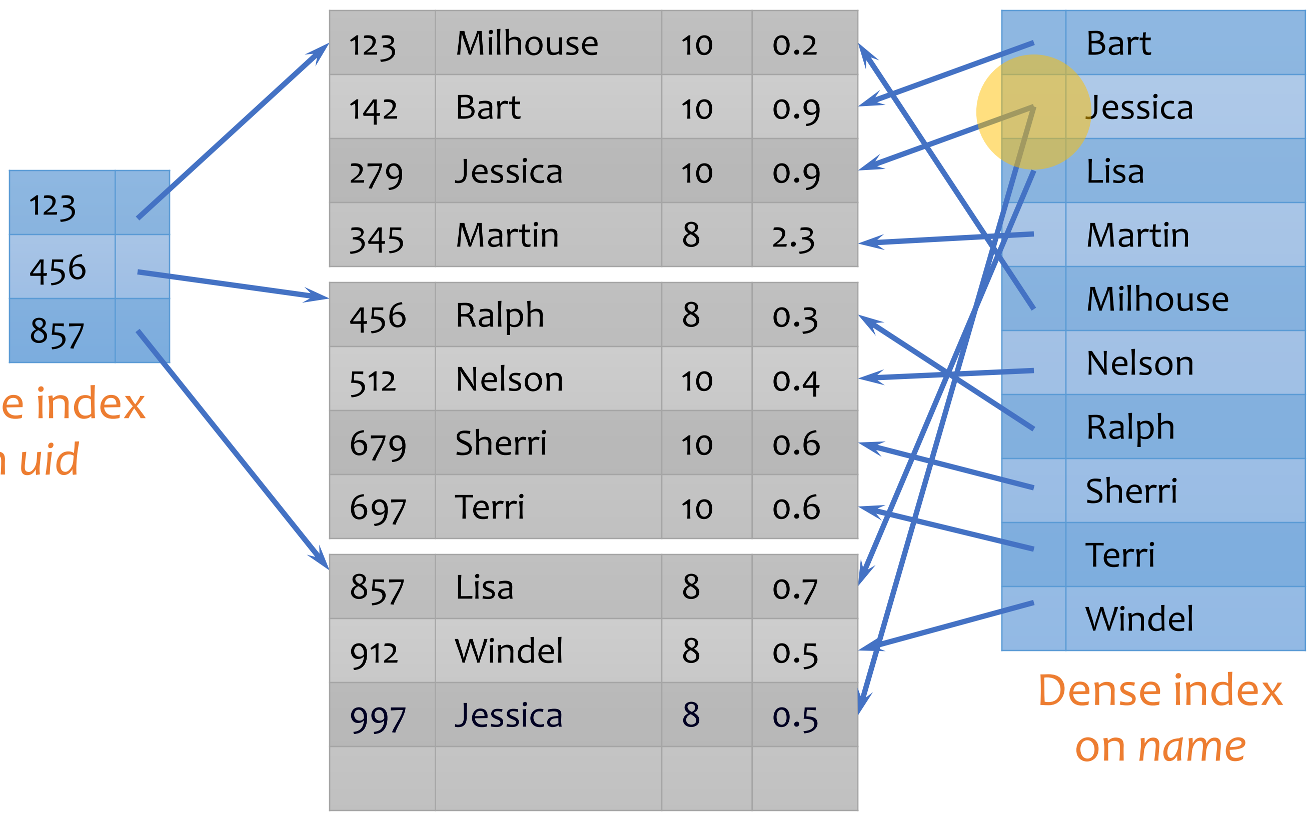
Dense v.s. Sparse indexes

- Dense: one index entry for each search key value
  - One entry may "point" to multiple records (e.g., two users named Jessica)



Dense v.s. Sparse indexes

- Sparse: one index entry for each block
  - Records must be clustered according to the search key on the disk

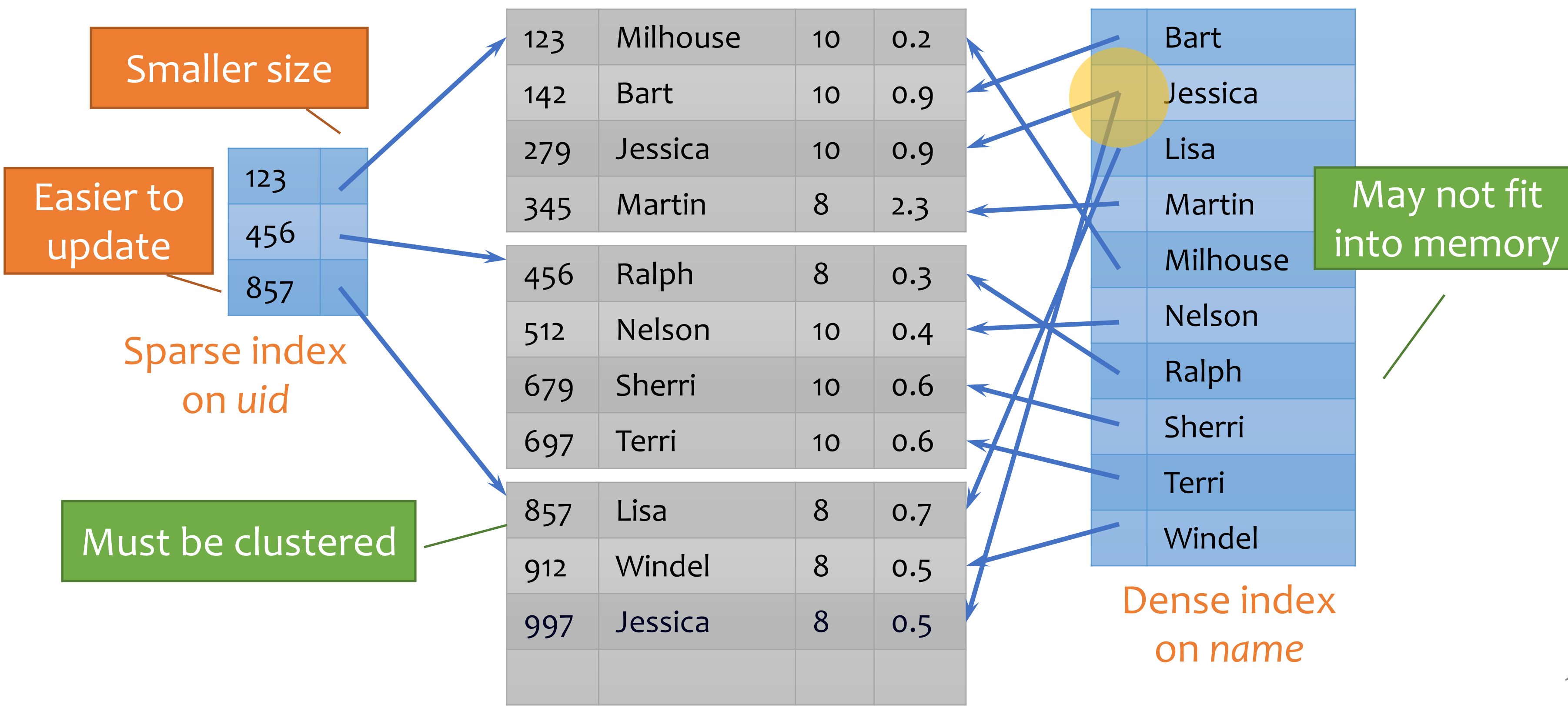


Sparse index on *uid*

Dense index on *name*

Dense v.s. Sparse indexes

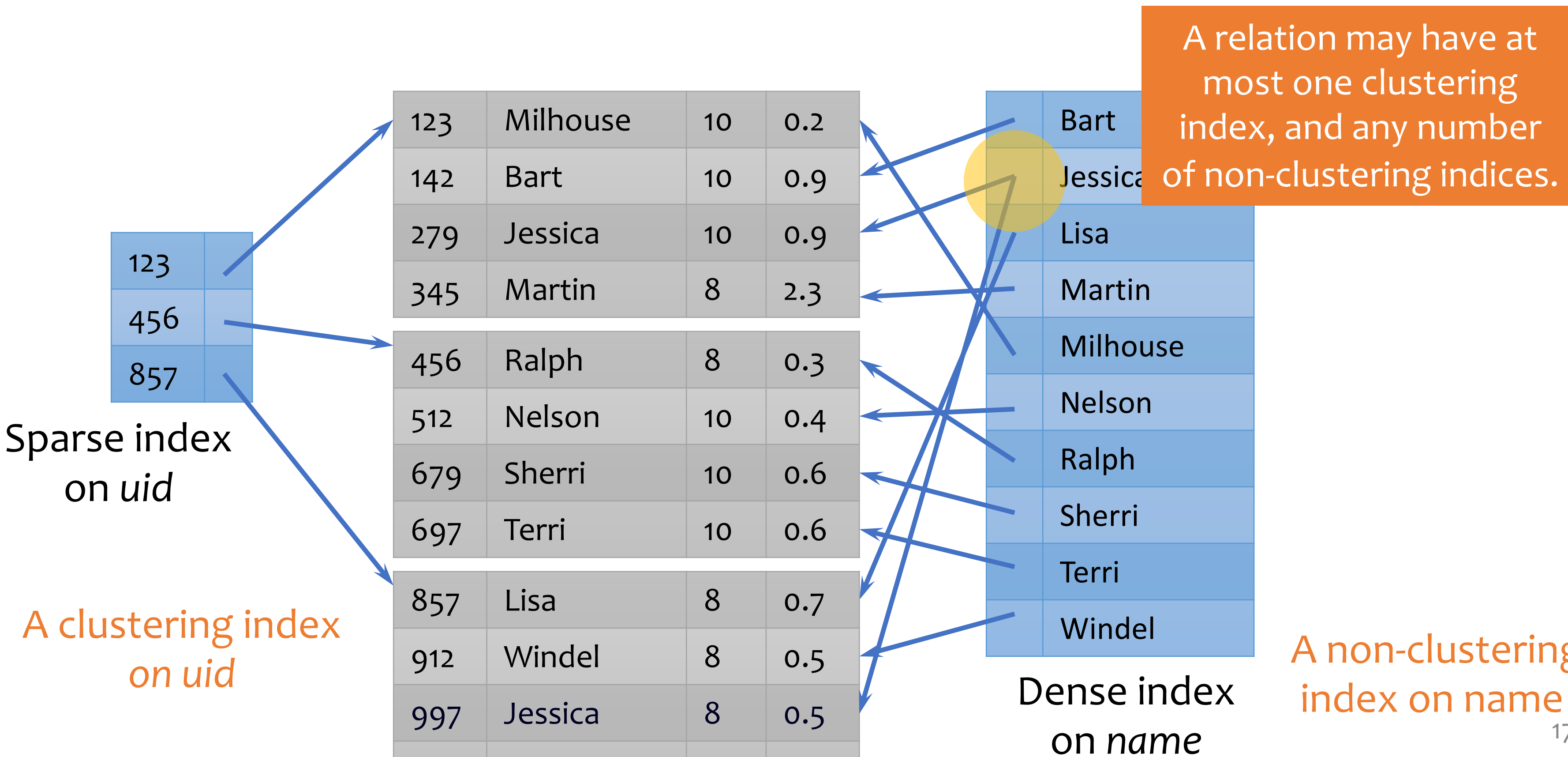
- Dense: one index entry for each search key value
- Sparse: one index entry for each block
  - Records must be clustered according to the search key



123	Milhouse	10	0.2
142	Bart	10	0.9
279	Jessica	10	0.9
345	Martin	8	2.3
456	Ralph	8	0.3
512	Nelson	10	0.4
679	Sherri	10	0.6
697	Terri	10	0.6
857	Lisa	8	0.7
912	Windel	8	0.5
997	Jessica	8	0.5

Clustering v.s. Non-Clustering indexes

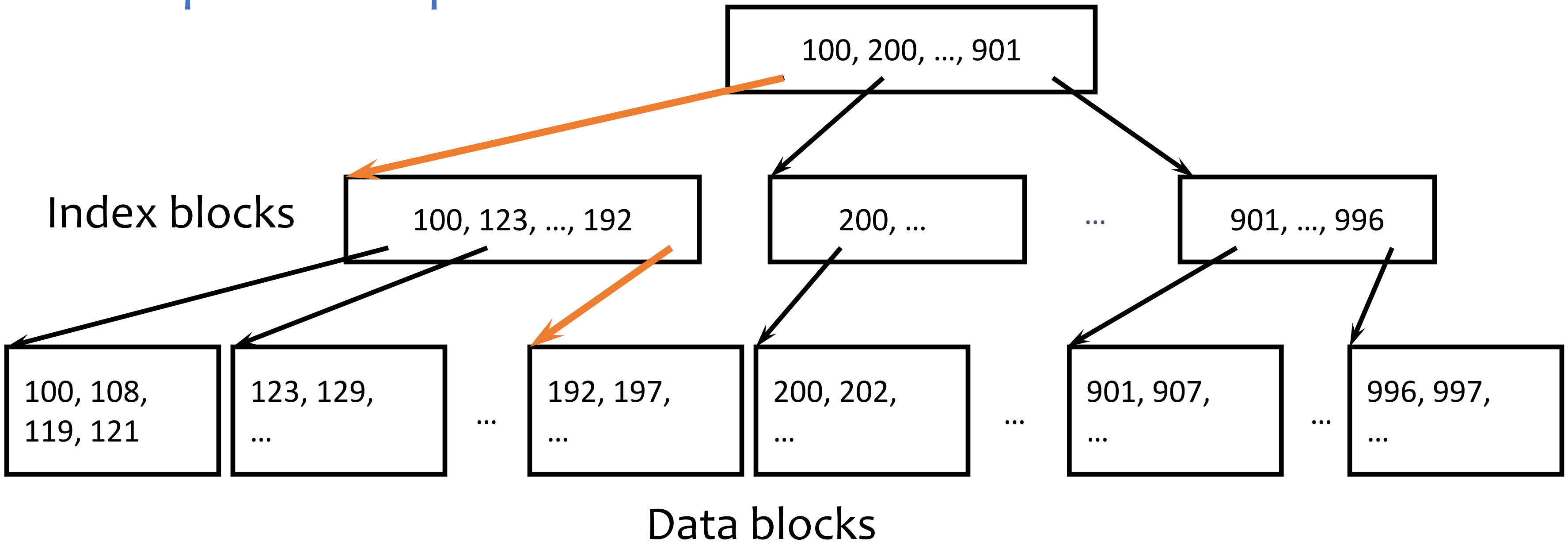
- An index on attribute A is a clustering index if tuples with similar A-values are stored together in the same block, and non-clustering otherwise.



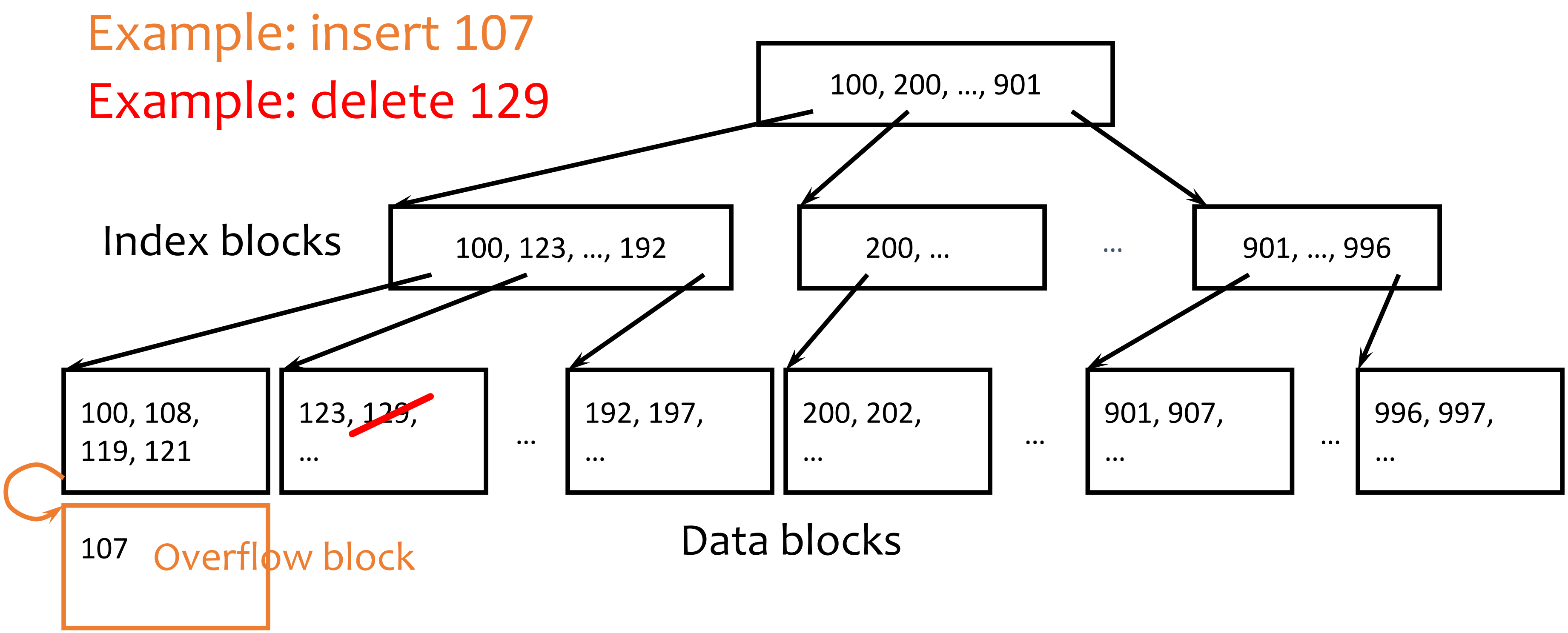
ISAM

- What if an index is still too big?
  - Put a another (sparse) index on top of that!
  - ☞ISAM (Index Sequential Access Method), more or less

Example: look up 197



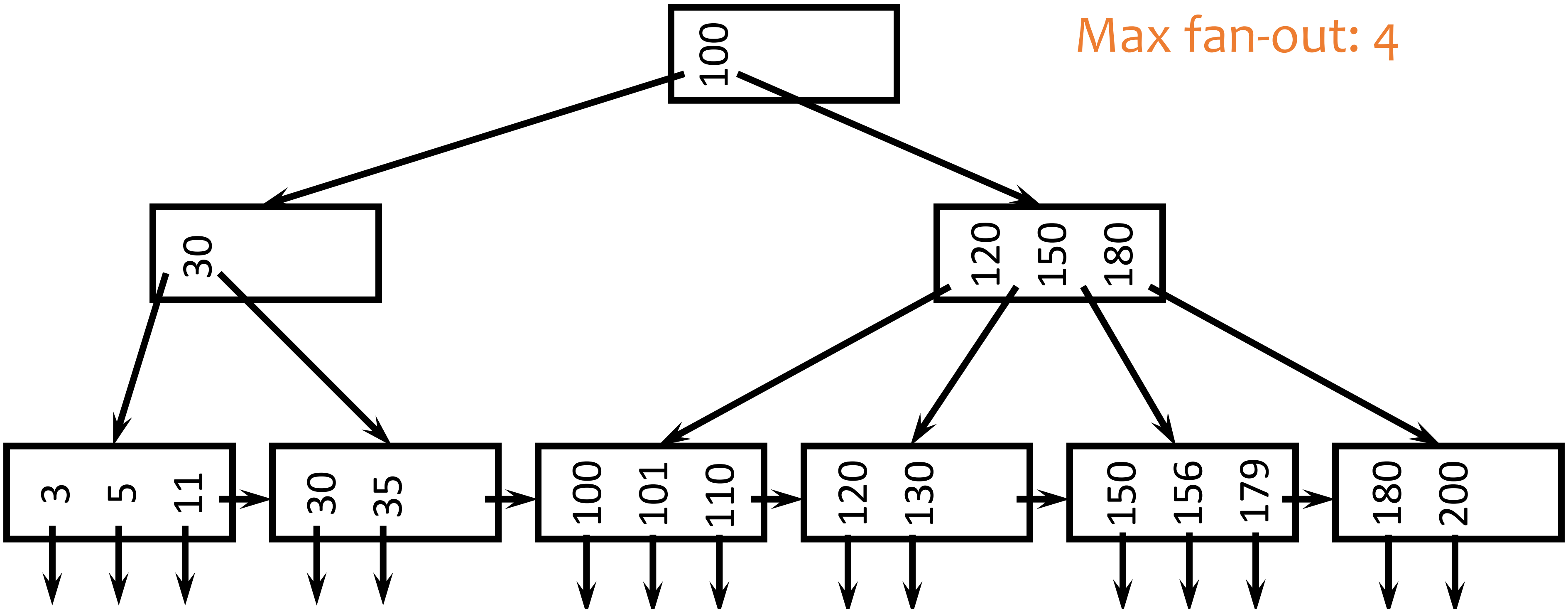
Updates with ISAM



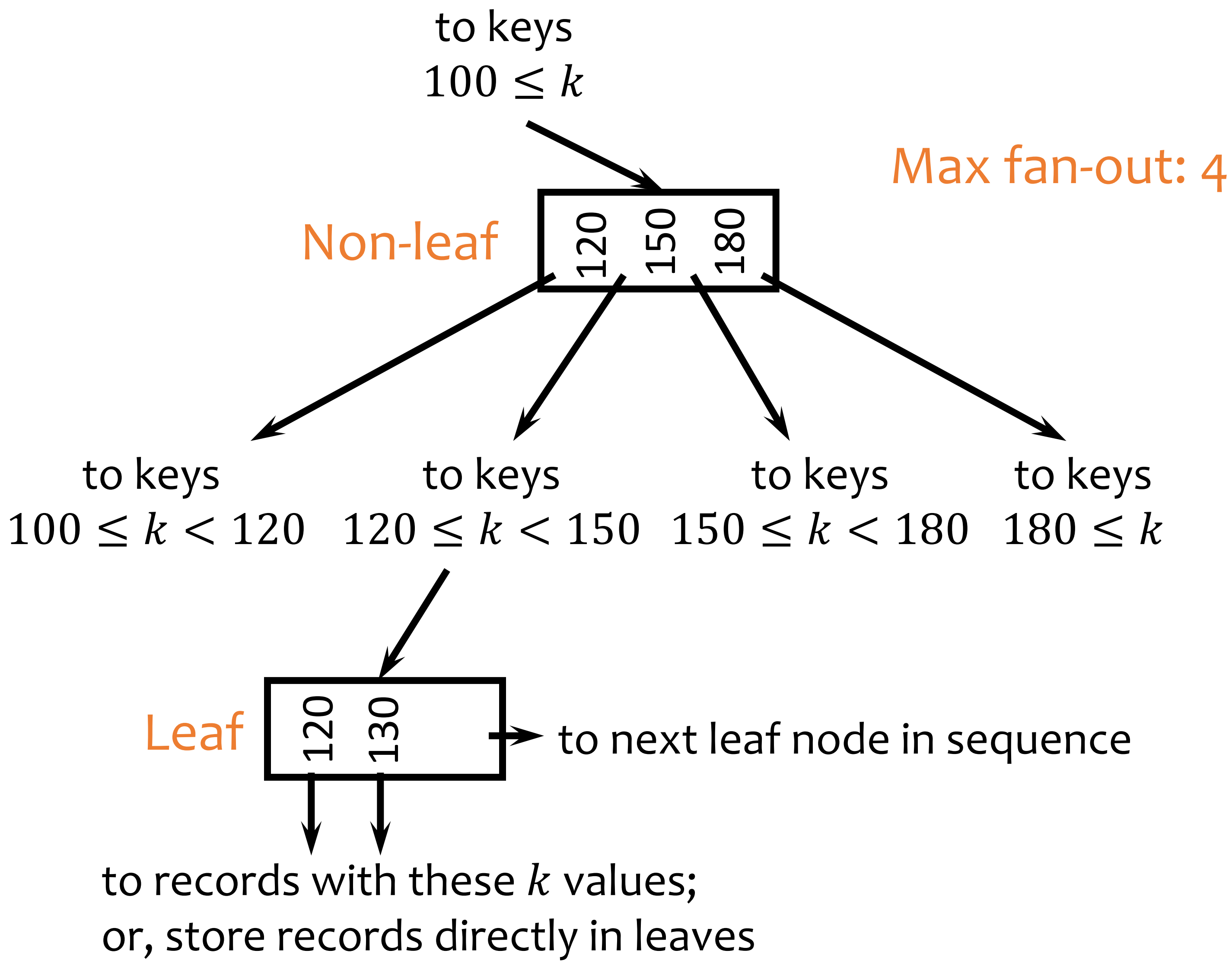
- Overflow chains and empty data blocks degrade performance
  - Worst case: most records go into one long chain, so lookups require scanning all data!

B⁺-tree

- A hierarchy of nodes with intervals
- Balanced: good performance guarantee
- Disk-based: one node per block; large fan-out

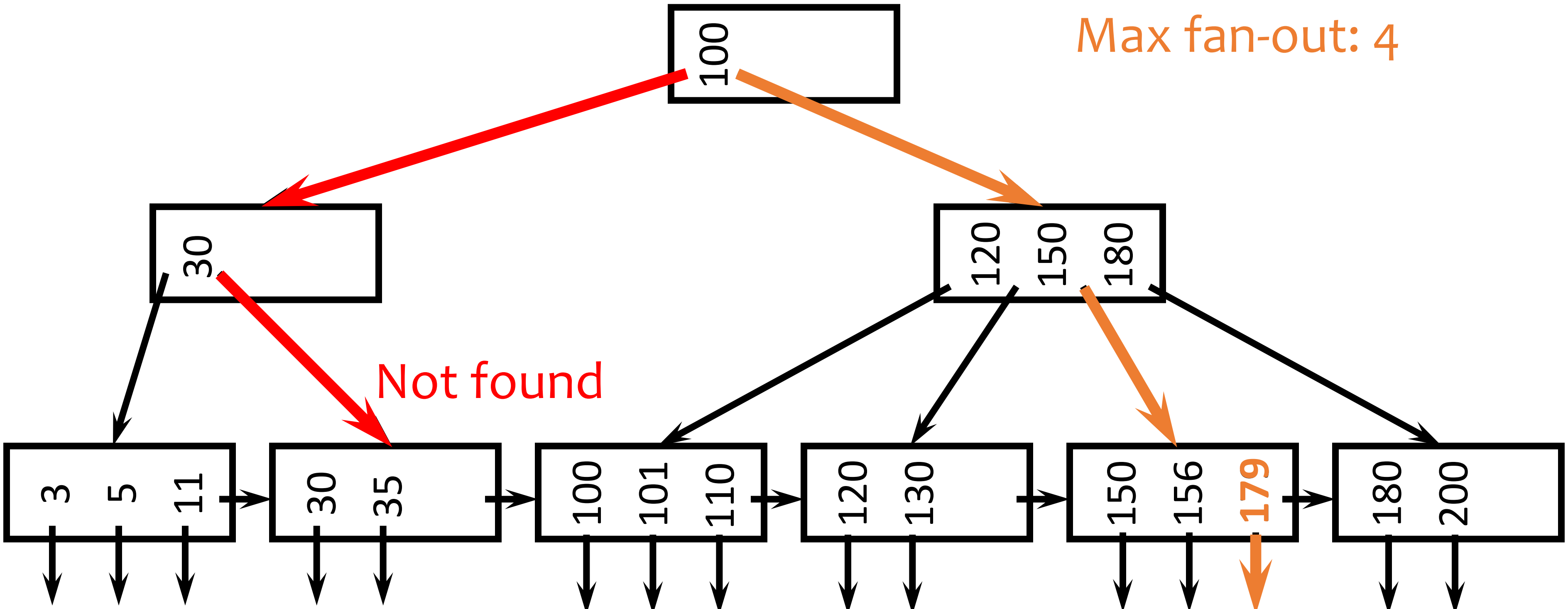


Sample B+-tree nodes



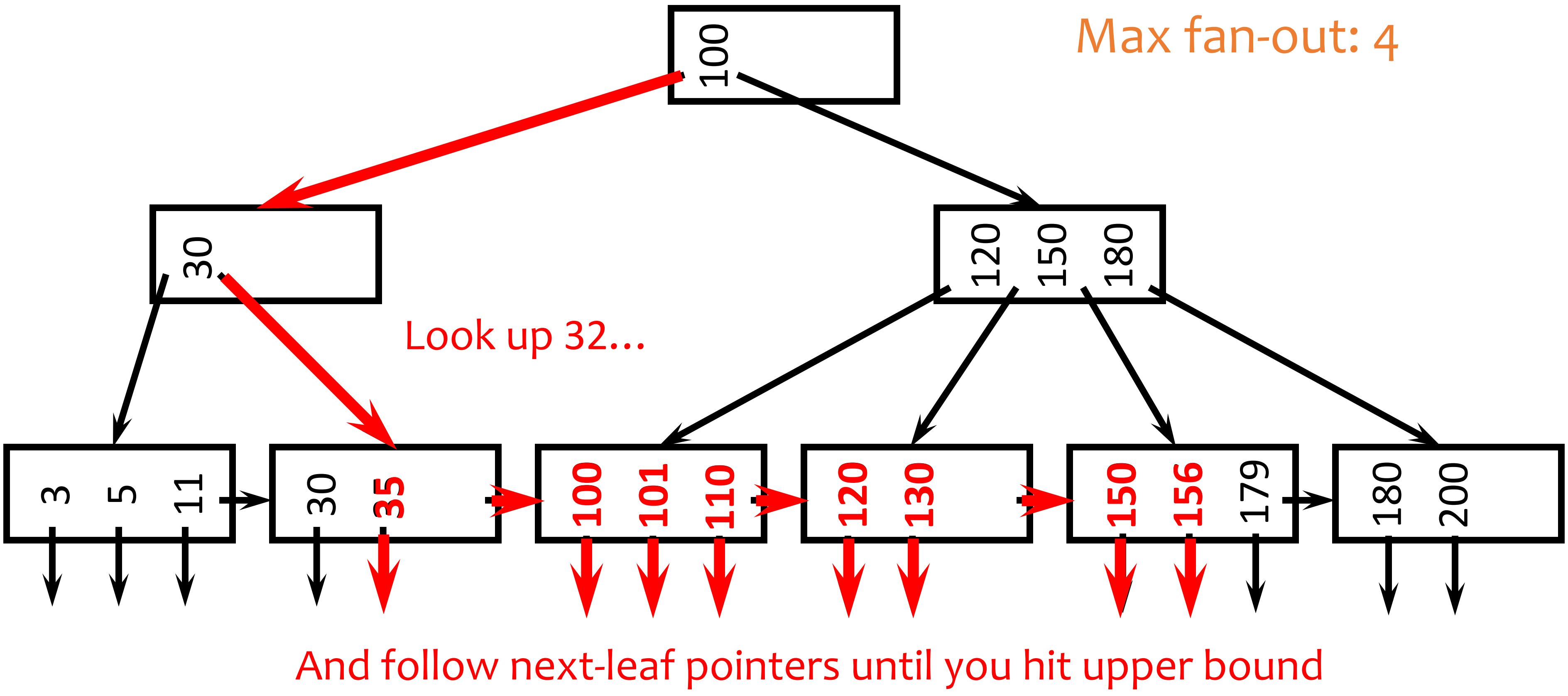
Lookups

- SELECT * FROM *R* WHERE *k* = 179;
- SELECT * FROM *R* WHERE *k* = 32;



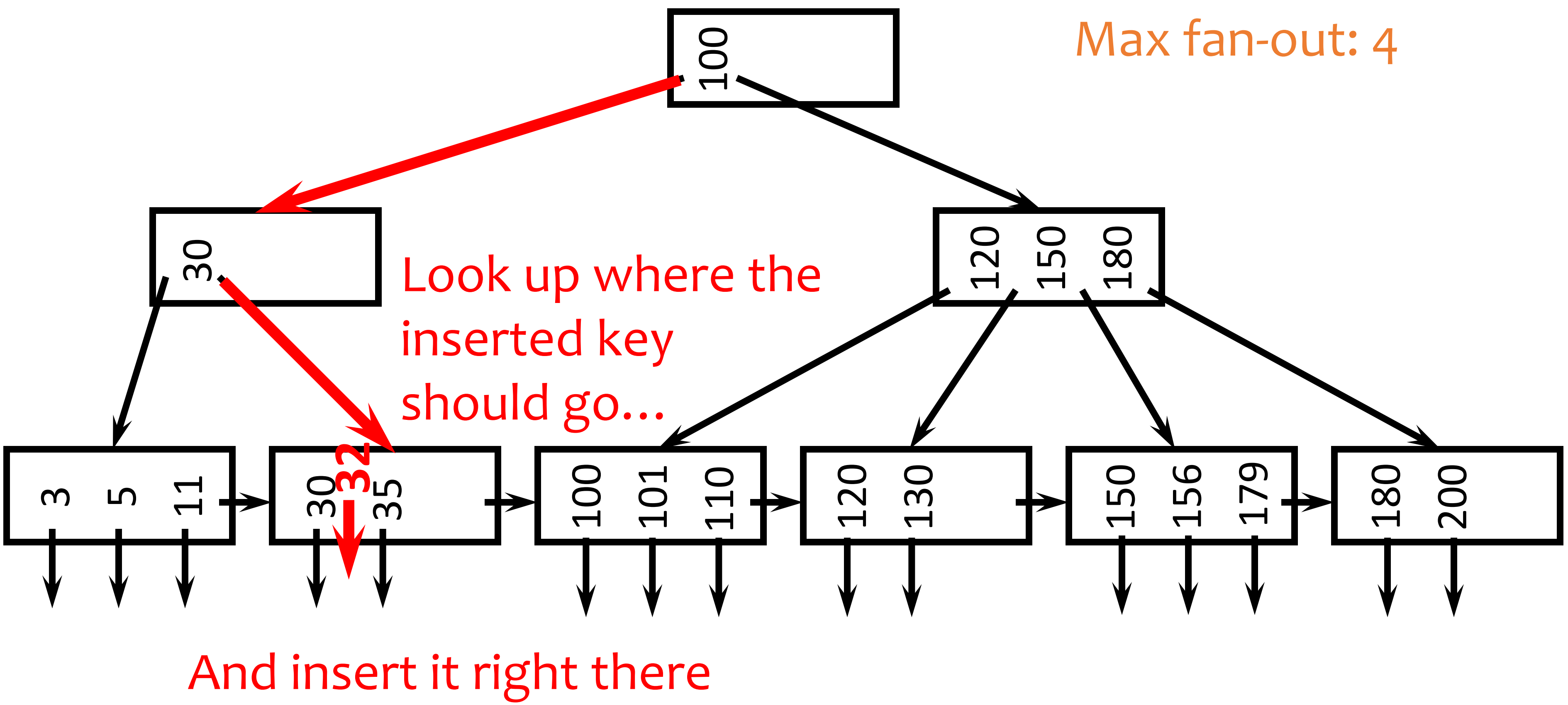
Range query

- SELECT * FROM *R* WHERE *k* > 32 AND *k* < 179;



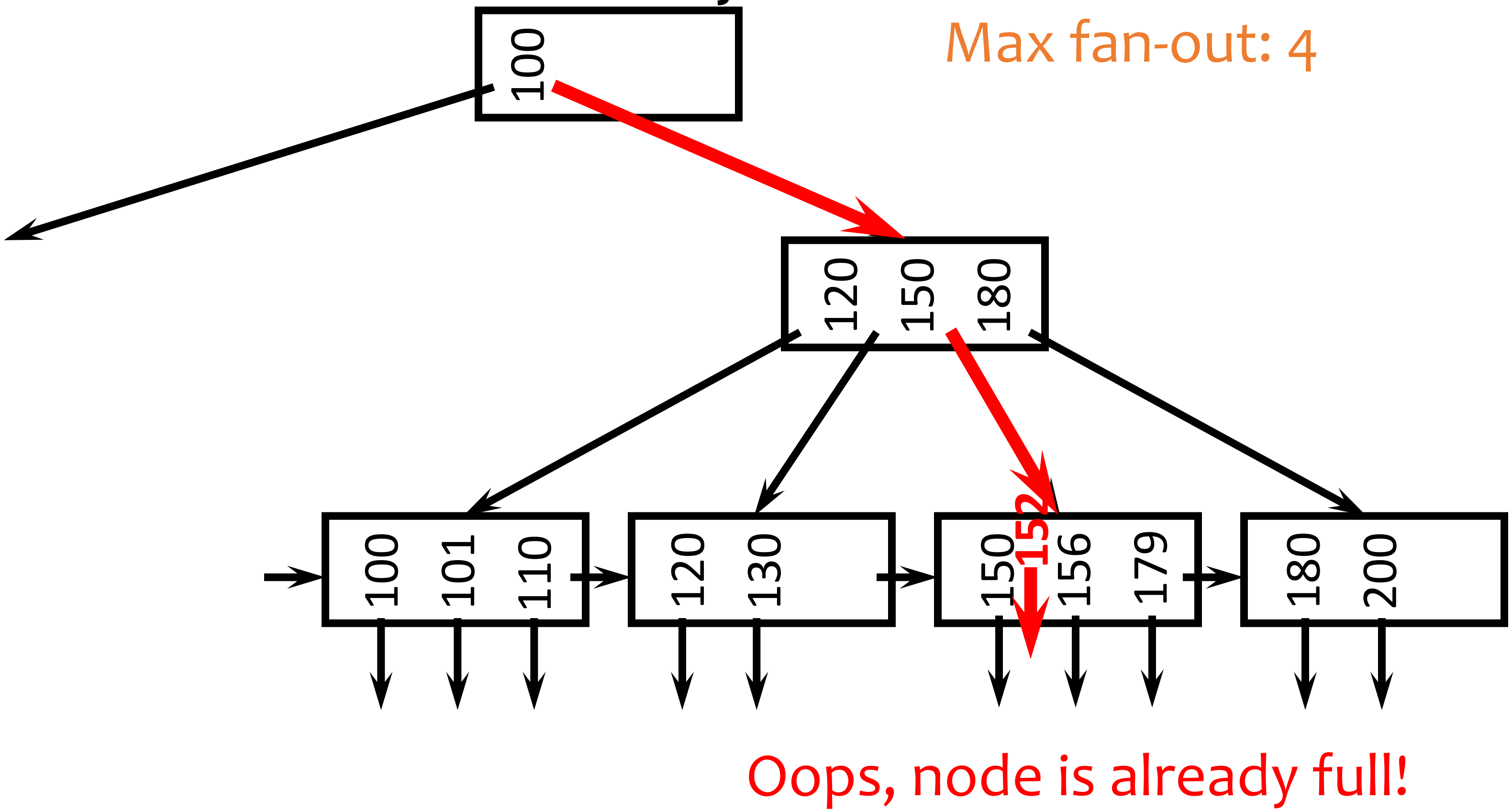
Insertion

- Insert a record with search key value 32

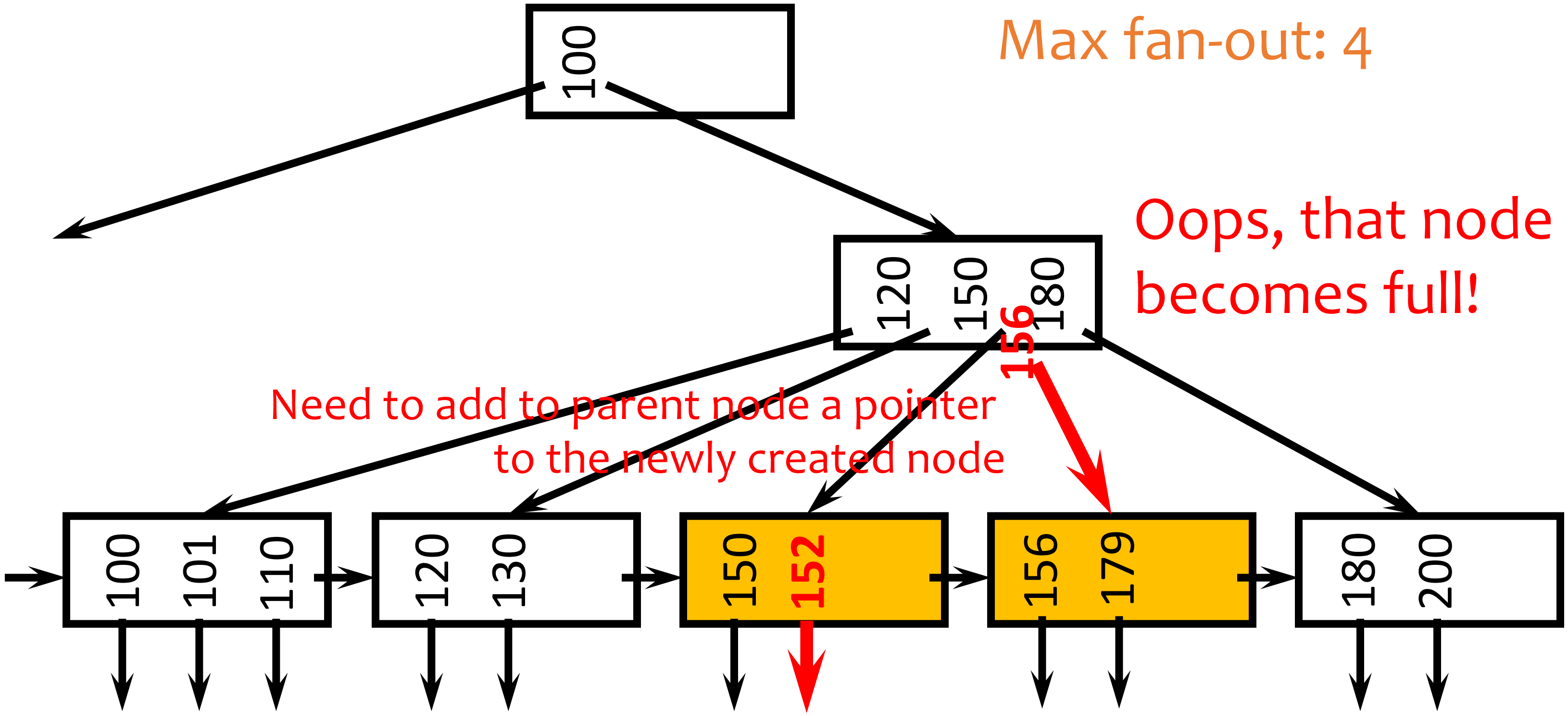


Another insertion example

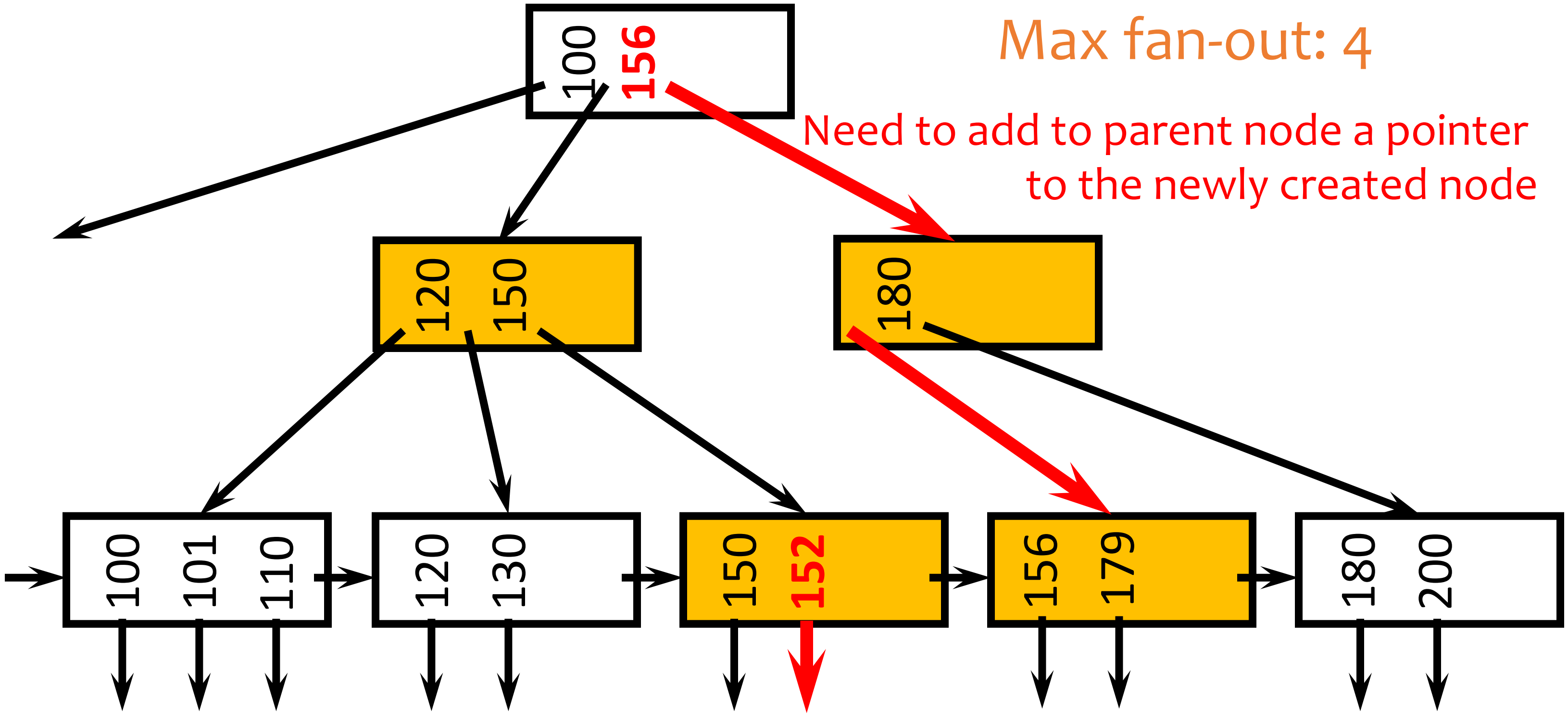
- Insert a record with search key value 152



Node splitting



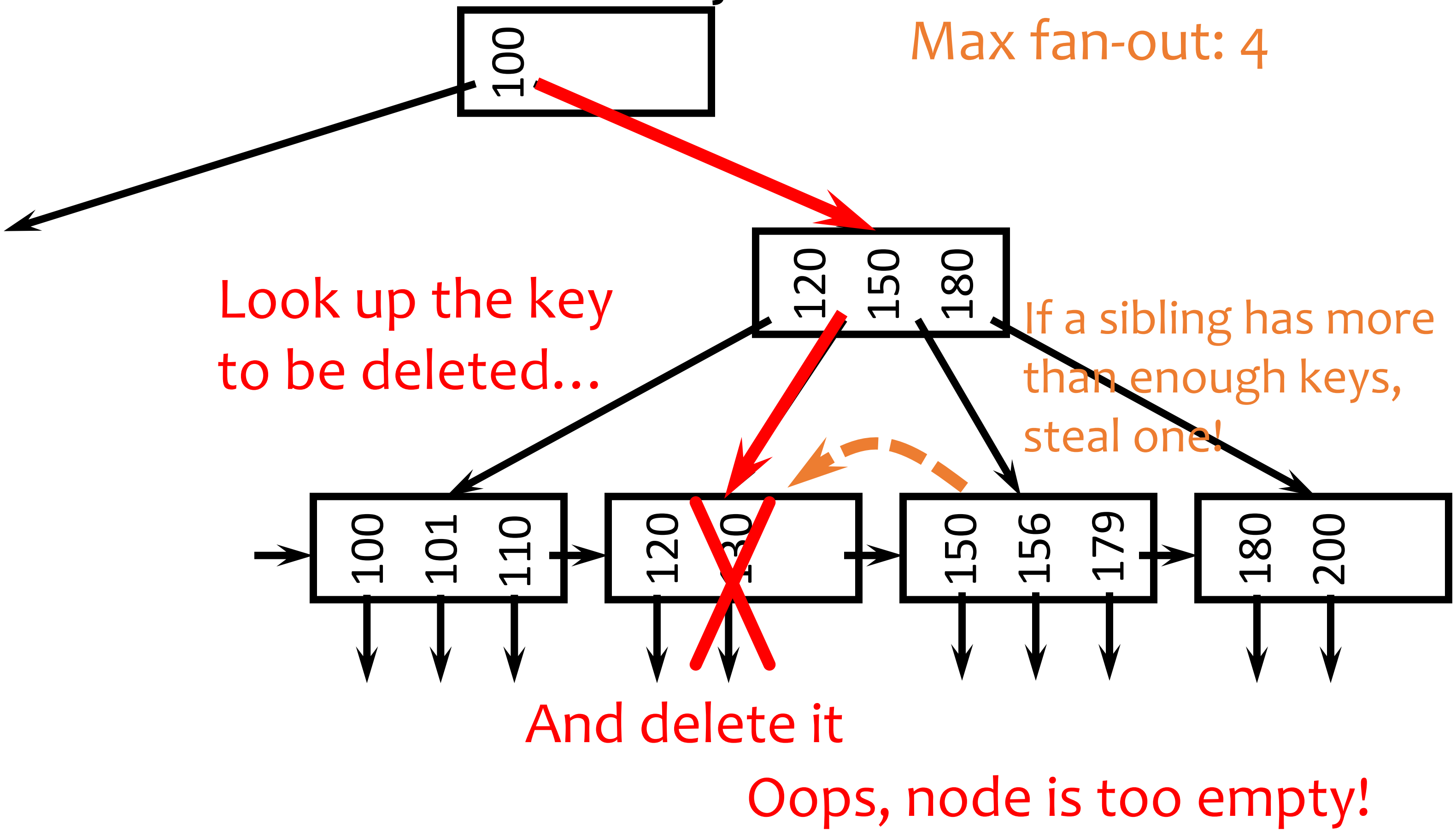
More node splitting



- In the worst case, node splitting can "propagate" all the way up to the root of the tree (not illustrated here)
  - Splitting the root introduces a new root of fan-out 2 and causes the tree to grow "up" by one level

Deletion

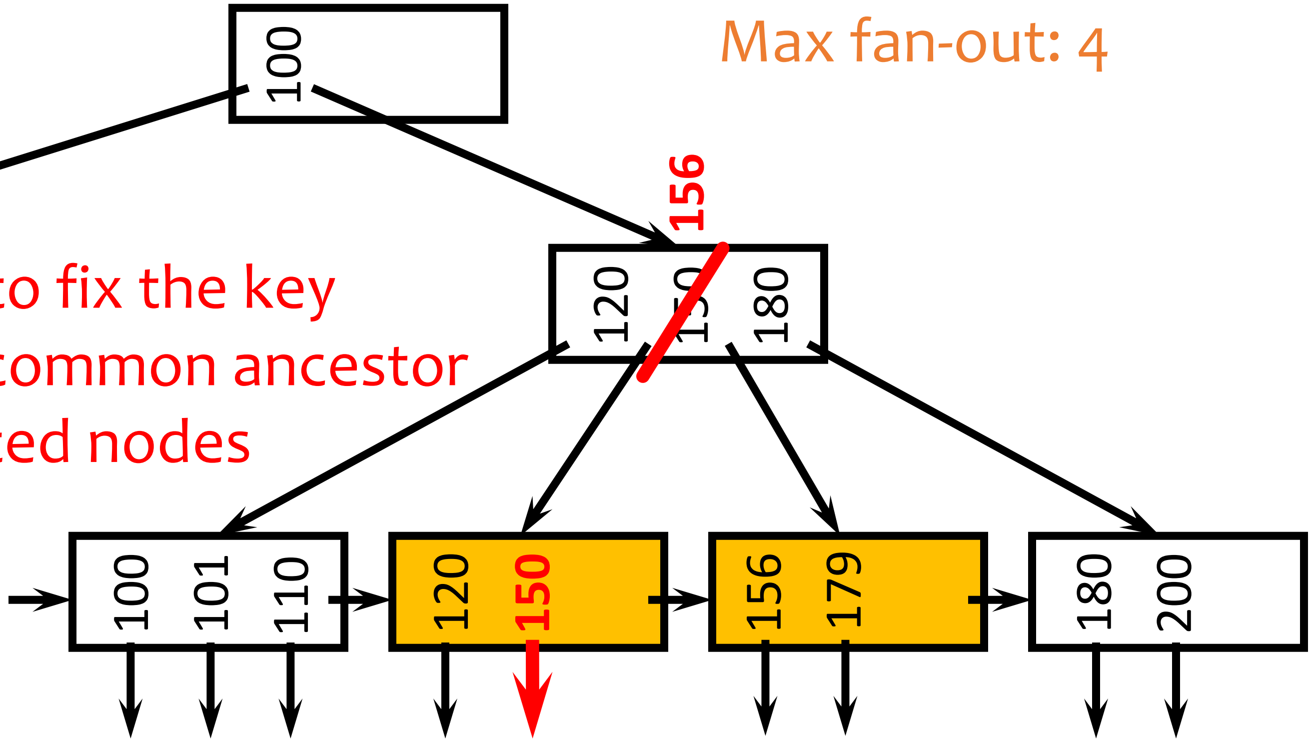
- Delete a record with search key value 130



Stealing from a sibling

Max fan-out: 4

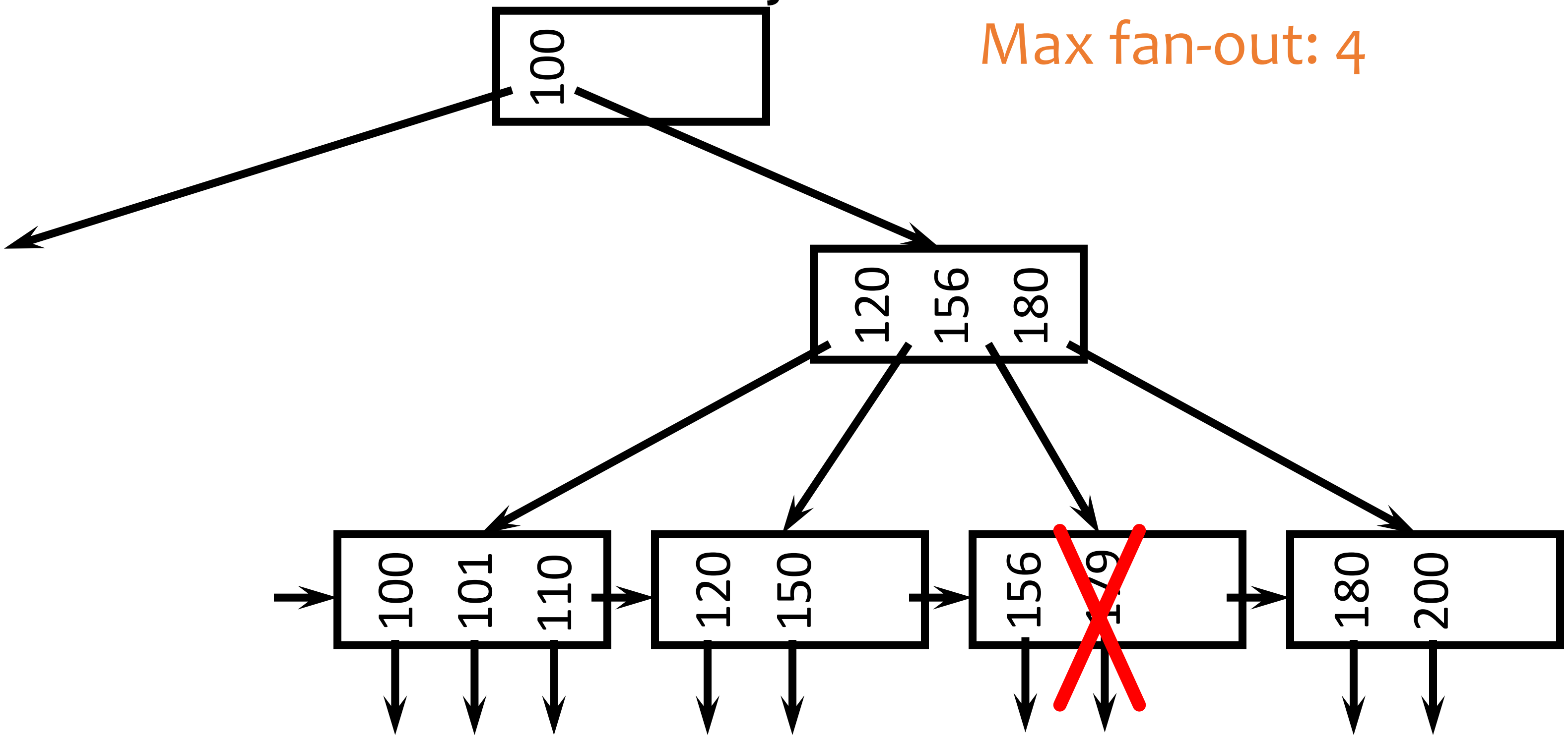
Remember to fix the key
in the least common ancestor
of the affected nodes



Another deletion example

- Delete a record with search key value 179

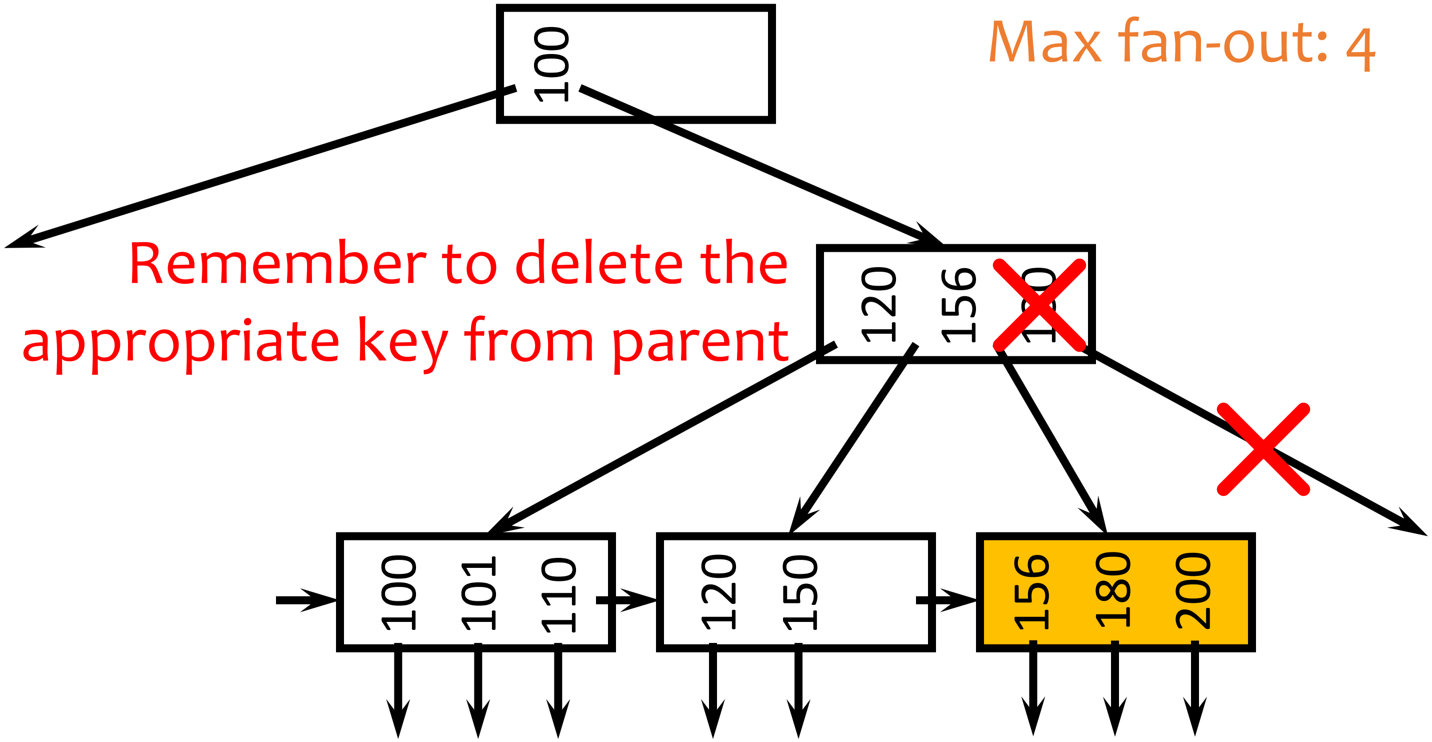
Max fan-out: 4



Cannot steal from siblings

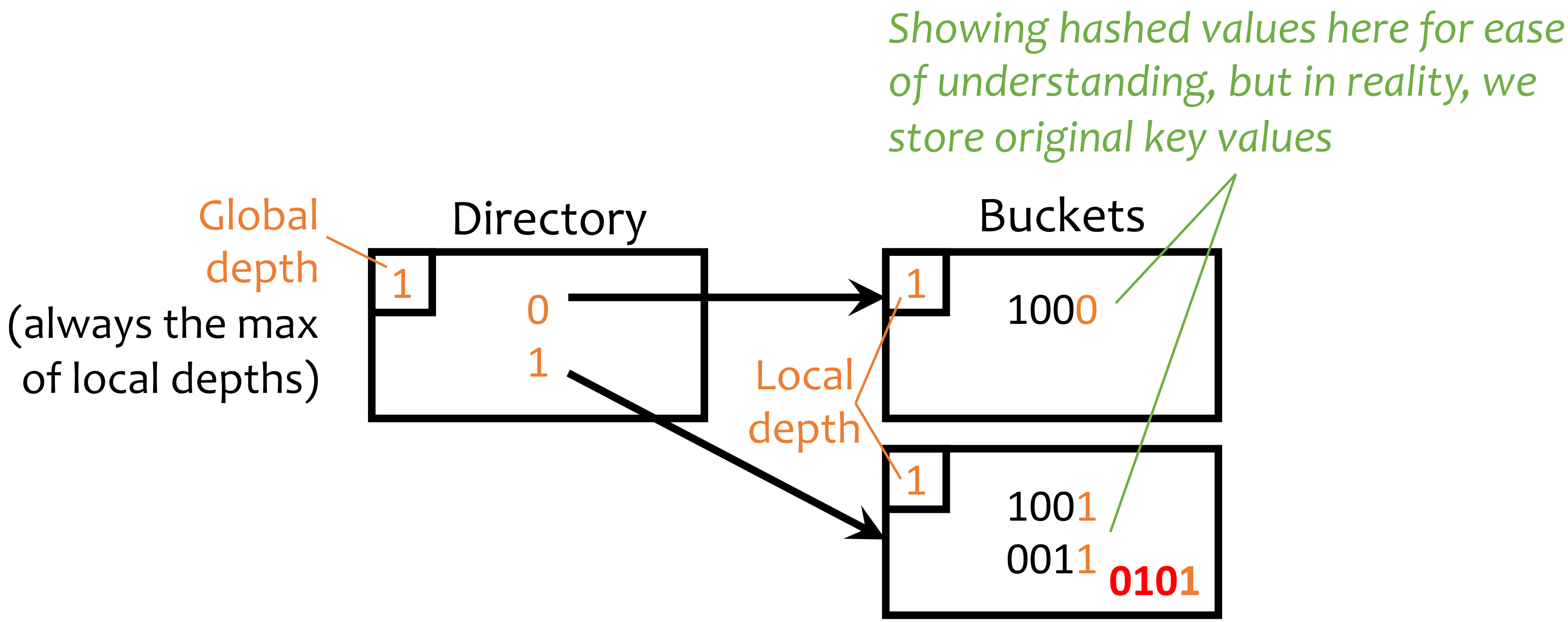
Then coalesce (merge) with a sibling!

Coalescing



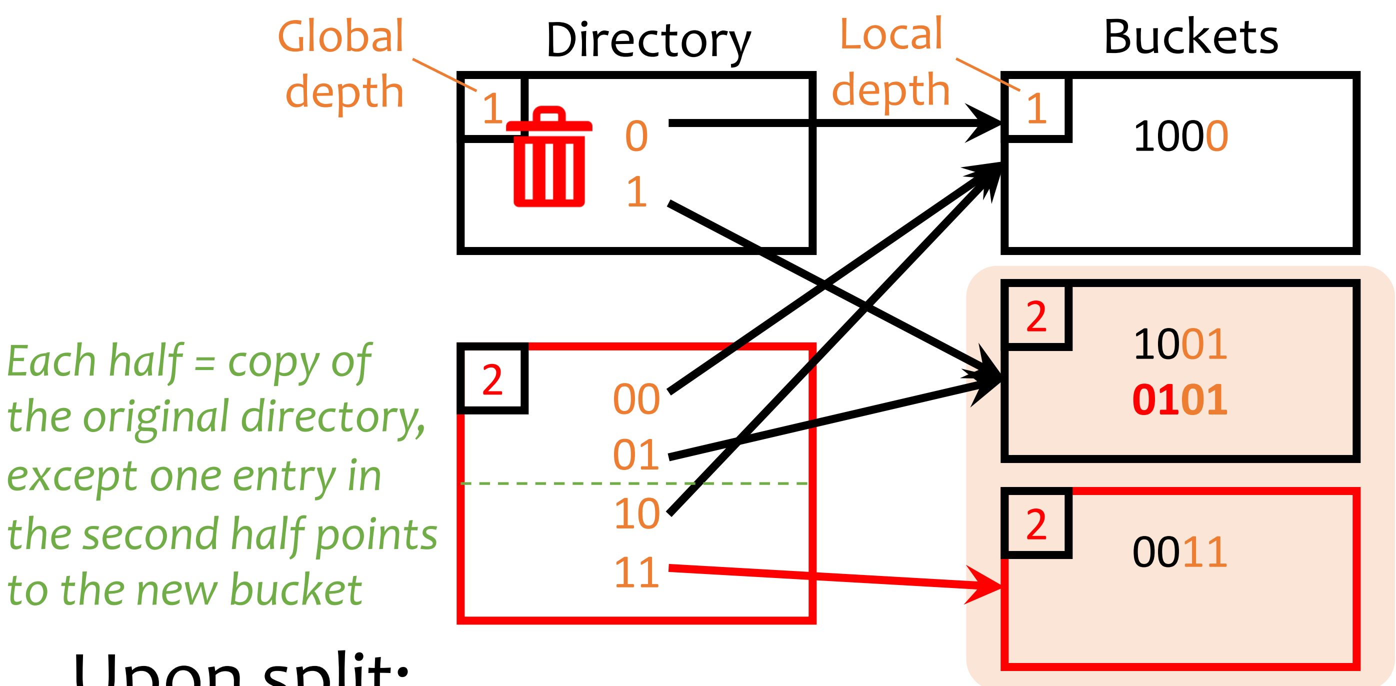
- Deletion can "propagate" all the way up to the root of the tree (not illustrated here)
  - When the root becomes empty, the tree "shrinks" by one level

Extensible hashing example – 1



- Insert k with $h(k) =$ **0101**
- Bucket too full? Split (next slide)
  - Allowing some overflow is also fine (and sometimes necessary)

Extensible hashing example – 2

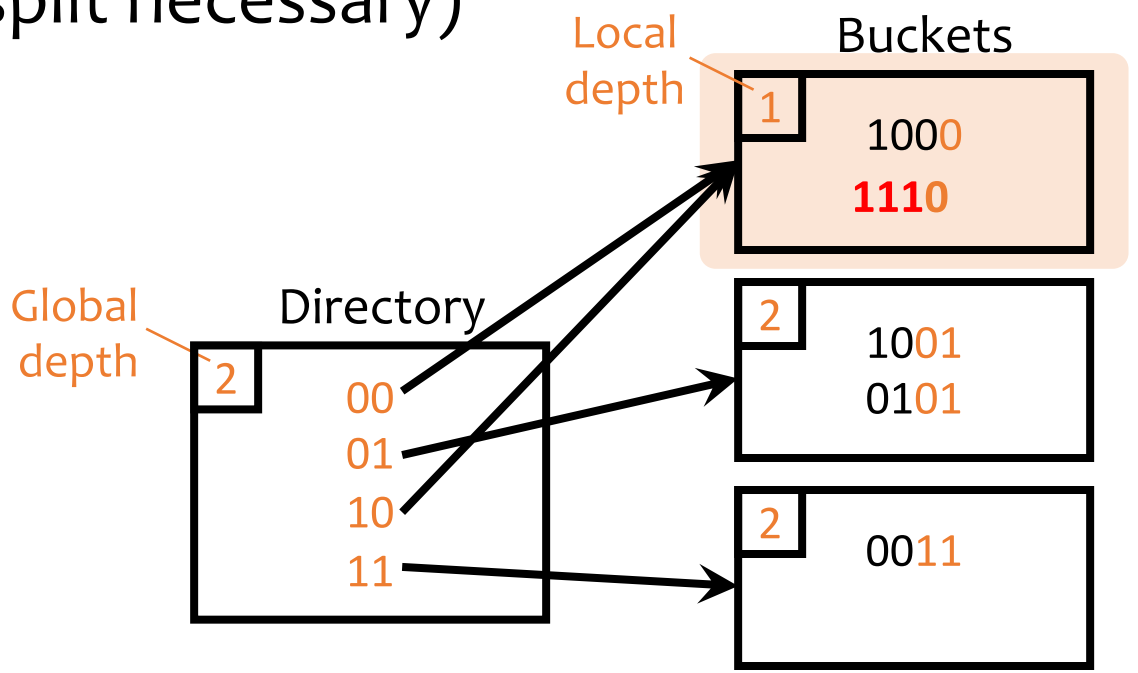


Upon split:

- ++local depth, redistribute contents, and ++global depth (double the directory size) if necessary

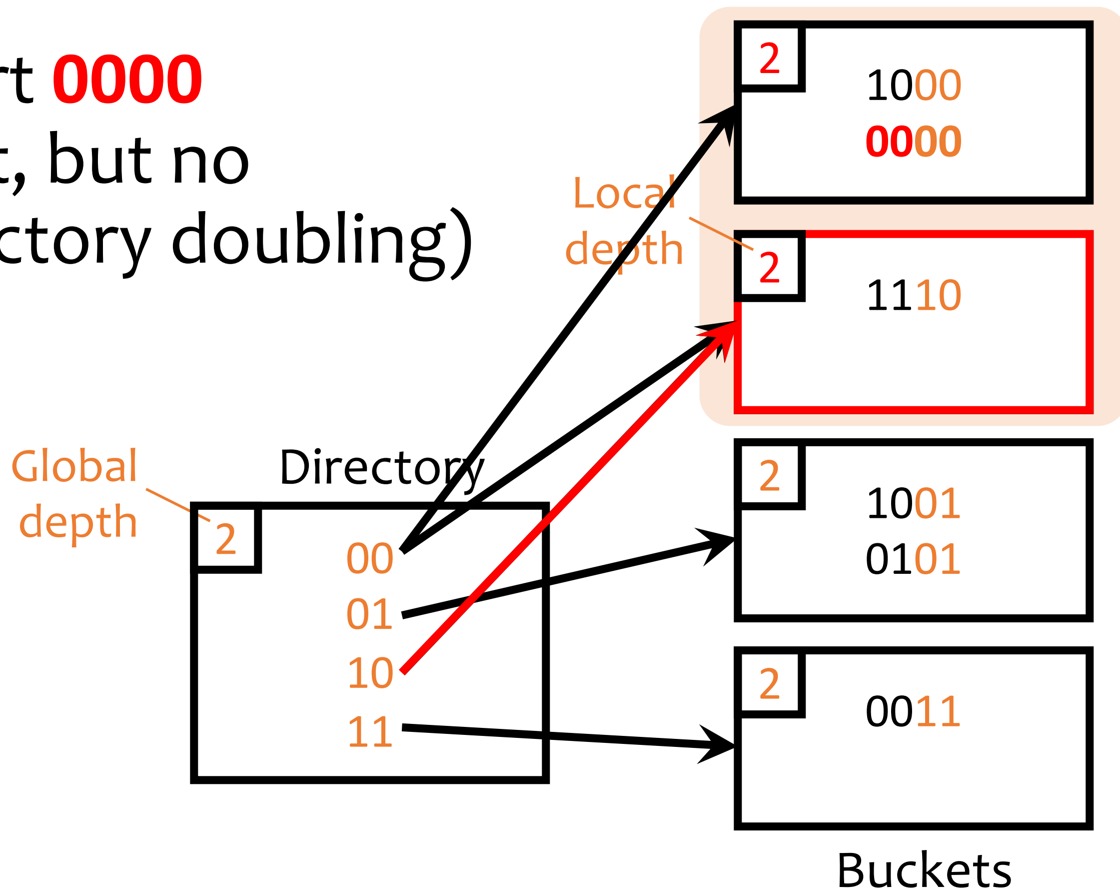
Extensible hashing example – 3

- Insert **1110**
  (no split necessary)



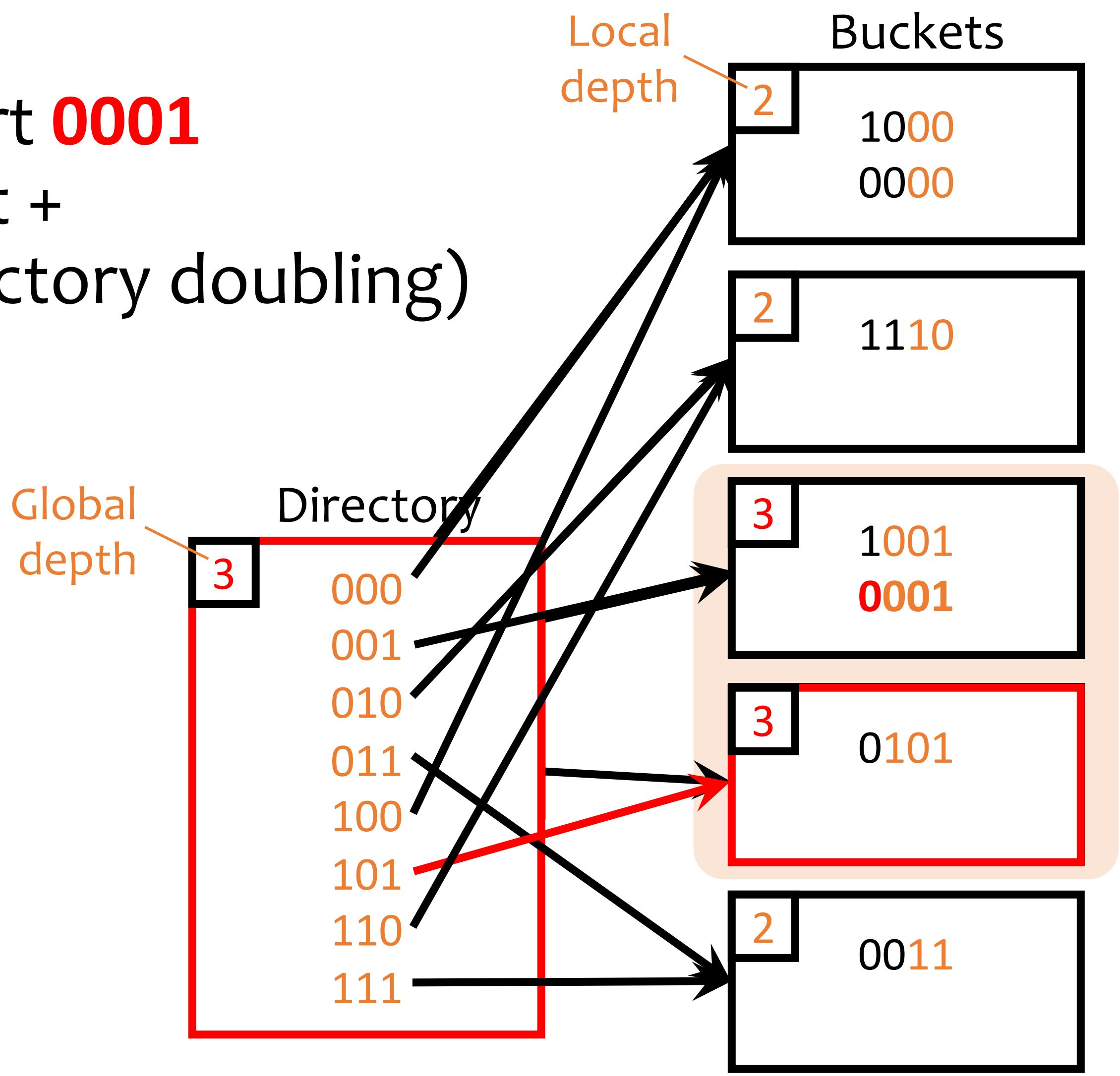
Extensible hashing example – 4

- Insert **0000** (split, but no directory doubling)



Extensible hashing example - 5

- Insert **0001** (split + directory doubling)



Linear hashing

- No extra indirection through a directory
  - Fix the splitting/growth order
  - Use some extra math to figure out the right bucket
- Grow only when utilization (avg. # entries per bucket / max # entries per block) exceeds a given threshold

n: # of primary buckets (not counting overflow blocks)
$i = \lceil \log_2 n \rceil$: # of hash bits in use (global depth)
threshold = 85% (a range of the buckets may use $i - 1$ bits)

$n = 2, i = 1$

bucket 0
0000
1010

bucket 1
1111

Local depth reflected by label, but no need to store explicitly

Linear hashing example – 1

$n = 2, i = 1$

bucket 0	bucket 1
0000	1111
1010	**0101**

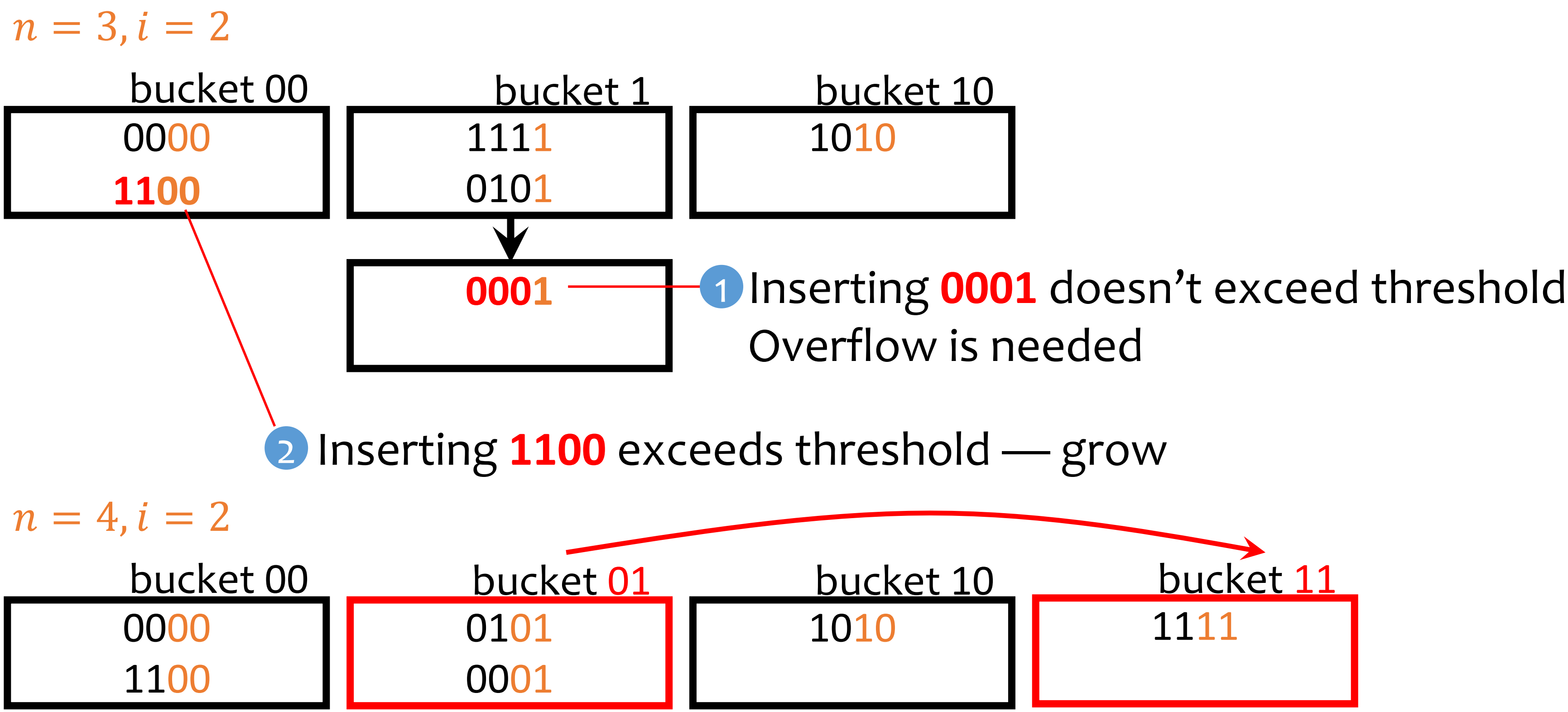
Inserting **0101** exceeds threshold — grow

- Split the first bucket with the lowest depth — it's always the bucket $n - 2^{\lfloor \log_2 n \rfloor}$ (0-based index)
  - Often not the bucket you are inserting into!
- File grows *linearly* at the end (hence the name)

$n = 3, i = 2$

bucket 00	bucket 1	bucket 10
0000	1111	1010
	0101	

Linear hashing example – 2



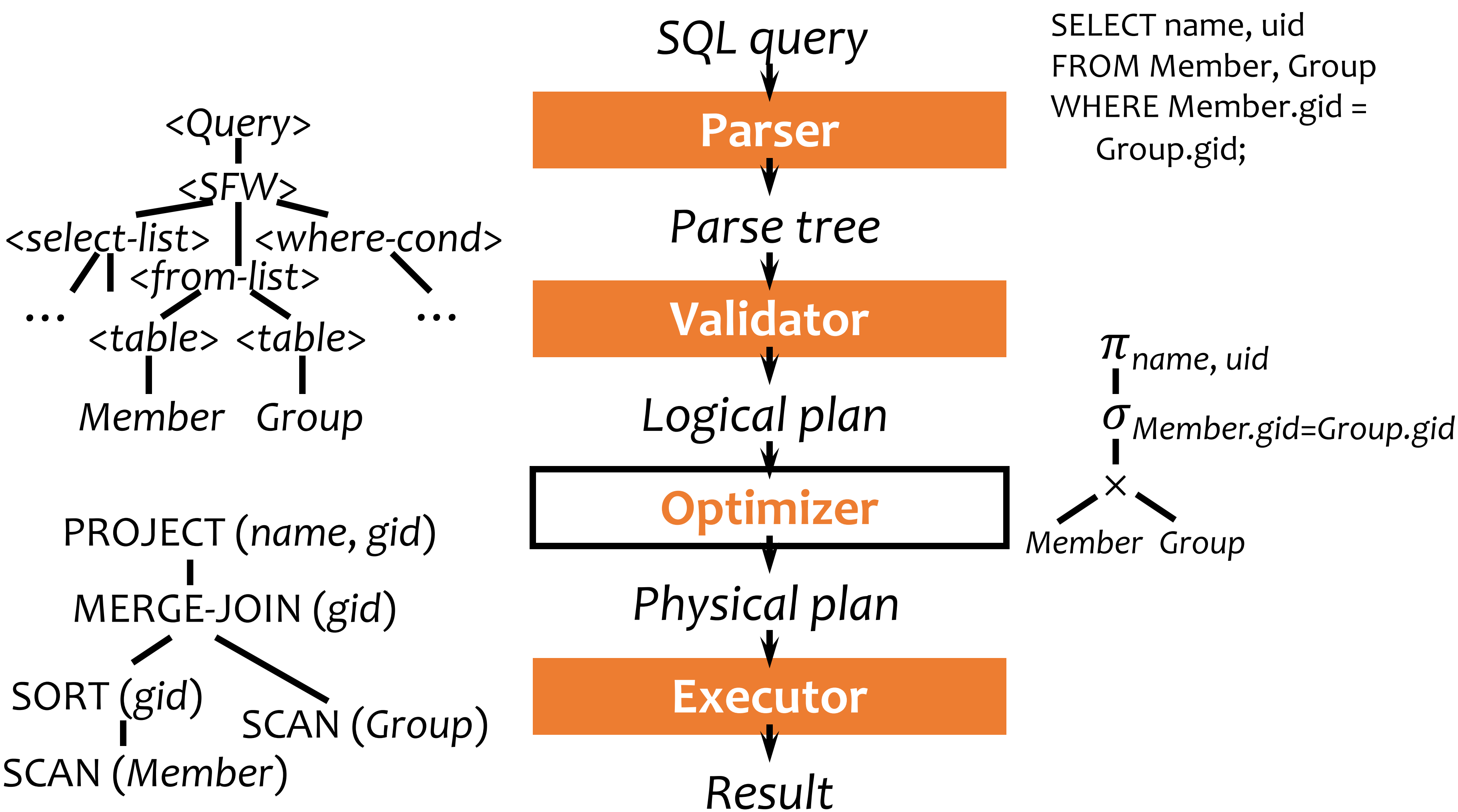
Index-only plan

- For example:
  - SELECT firstname, pop FROM User WHERE pop > ‘0.8’ AND firstname = ‘Bob’;
  - non-clustering index on (firstname, pop)

- A (non-clustered) index contains all the columns needed to answer the query without having to access the tuples in the base relation.
  - Avoid one disk I/O per tuple
  - The index is much smaller than the base relation

Query Processing & Optimization

A query’s trip through the DBMS



Query execution

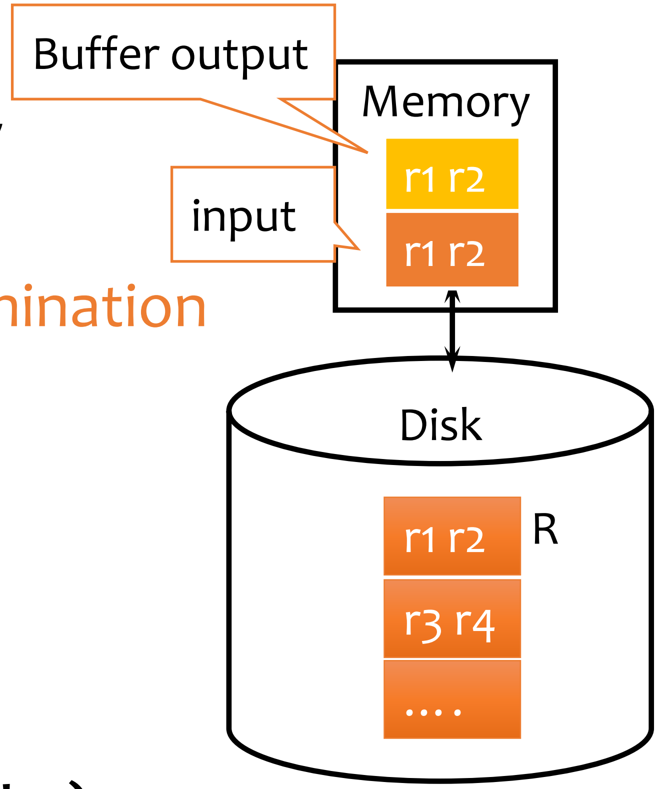
- Scan
  - Table scan
  - Selection, Duplicate-preserving projection
  - Nested-loop join
- Sort
  - External merge sort
  - Duplicate elimination, Grouping and Aggregation
  - Sort-merge join, Union (set), Difference, Intersection
- Hash
  - Hash join, union (set), difference, intersection, duplicate elimination, grouping and aggregation
- Index
  - Selection, index nested-loop join, zig-zag join

Notation and Assumption

- Relations: R, S
- Tuples: r, s
- Number of tuples: $|R|$, $|S|$
- Number of disk blocks: $B(R)$, $B(S)$
- Number of memory blocks available: M
- Cost metric
  - Number of I/O's (blocks transferred between memory and disk)
  - Memory requirement
- Not counting the cost of writing the result out
  - Same for any algorithm
  - Maybe not needed – results may be pipelined into downstream operator

Table scan

- Scan table R and process the query
  - Selection over R
  - Projection of R without duplicate elimination
- I/O's: $B(R)$
  - Stop early if it is a lookup by key
- Memory requirement: $M \geq 2$ (blocks)
  - 1 for input, 1 for buffer output
  - Increase M does not improve I/O



Tuple-based Nested-loop join

$R \bowtie_p S$

- For each block of R, and for each r in the block:
  For each block of S, and for each s in the block:
  Output rs if p evaluates to true over r and s

- R is called the outer table; S is called the inner table
- I/O's: $B(R) + |R| \cdot B(S)$

Blocks of R are moved into memory only once

Blocks of S are moved into memory $|R|$ times

- Memory requirement: 3

Block-based nested-loop join

$R \bowtie_p S$

- For each block of R

  For each block of S

  For each r in the R block

  For each s in the S block

  Output rs if p evaluates to true over r and s

- I/O's: $B(R) + B(R) \cdot B(S)$

Blocks of R are moved into memory only once

Blocks of S are moved into memory B(R) times

- Memory requirement: same as before

More improvements

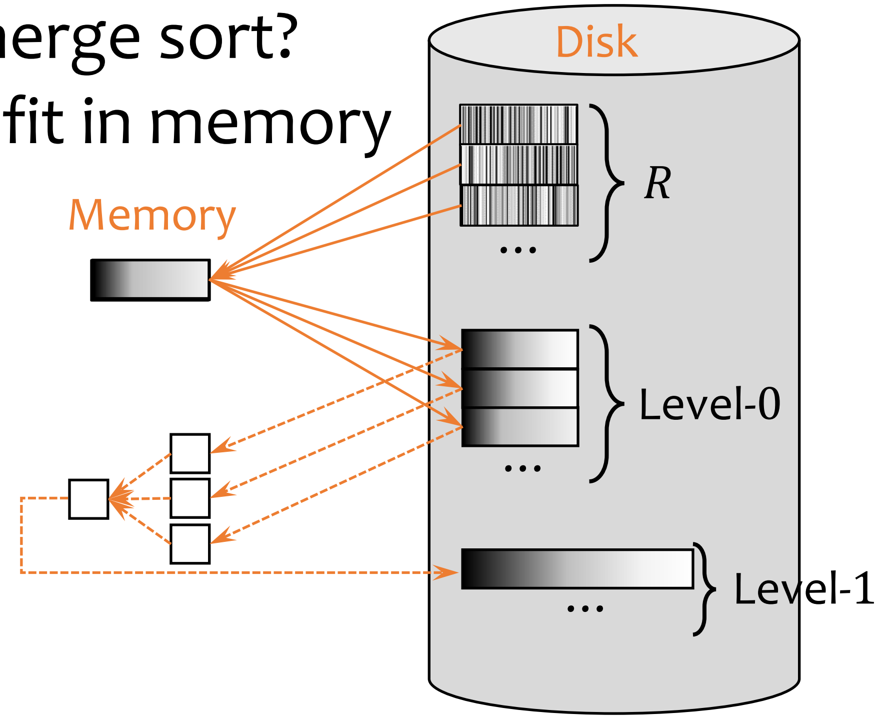
- Stop early if the key of the inner table is being matched
- Make use of available memory
  - Stuff memory with as much of R as possible, stream S by, and join every S tuple with all R tuples in memory
  - I/O's: $B(R) + \left\lceil \frac{B(R)}{M-2} \right\rceil \cdot B(S)$
    - Or, roughly: $B(R) \cdot B(S)/M$
  - Memory requirement: M (as much as possible)
- Which table would you pick as the outer? (exercise)

External merge sort

Remember (internal-memory) merge sort?

Problem: sort R, but R does not fit in memory

- Pass 0: read M blocks of R at a time, sort them, and write out a level-0 run



- Pass 1: merge $(M-1)$ level-0 runs at a time, and write out a level-1 run
- Pass 2: merge $(M-1)$ level-1 runs at a time, and write out a level-2 run

…

- Final pass produces one sorted run

Sort-merge join

$R \bowtie_{R.A=S.B} S$

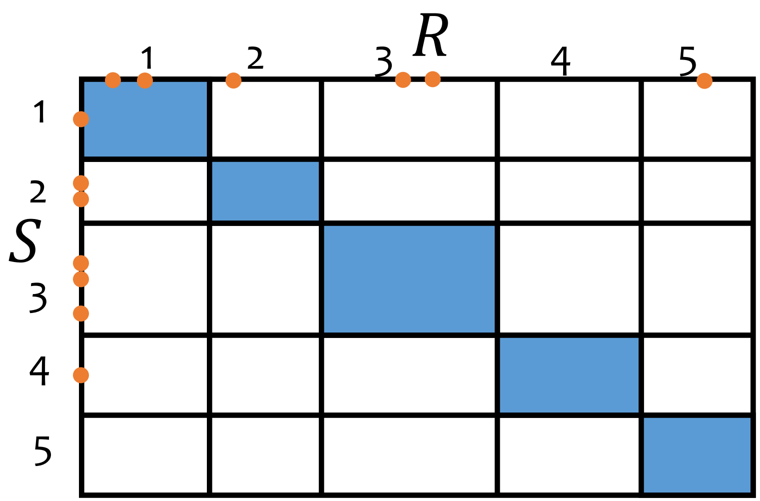
- Sort R and S by their join attributes
- $r, s \leftarrow$ the first tuples in sorted R and S
- Repeat until one of R and S is exhausted:
  - If $r.A > s.B$, then $s \leftarrow$ next tuple in S
  - Else if $r.A < s.B$, then $r \leftarrow$ next tuple in R
  - Else $(r.A = s.B)$

    output all matching tuples;
    $r, s \leftarrow$ next tuples in R and S respectively
- If R is not exhausted, output remaining tuples in R
- If S is not exhausted, output remaining tuples in S

Hash join

$R \bowtie_{R.A=S.B} S$

- Main idea
  - Partition R and S by hashing their join attributes, and then consider corresponding partitions of R and S
  - If $r.A$ and $s.B$ get hashed to different partitions, they don't join

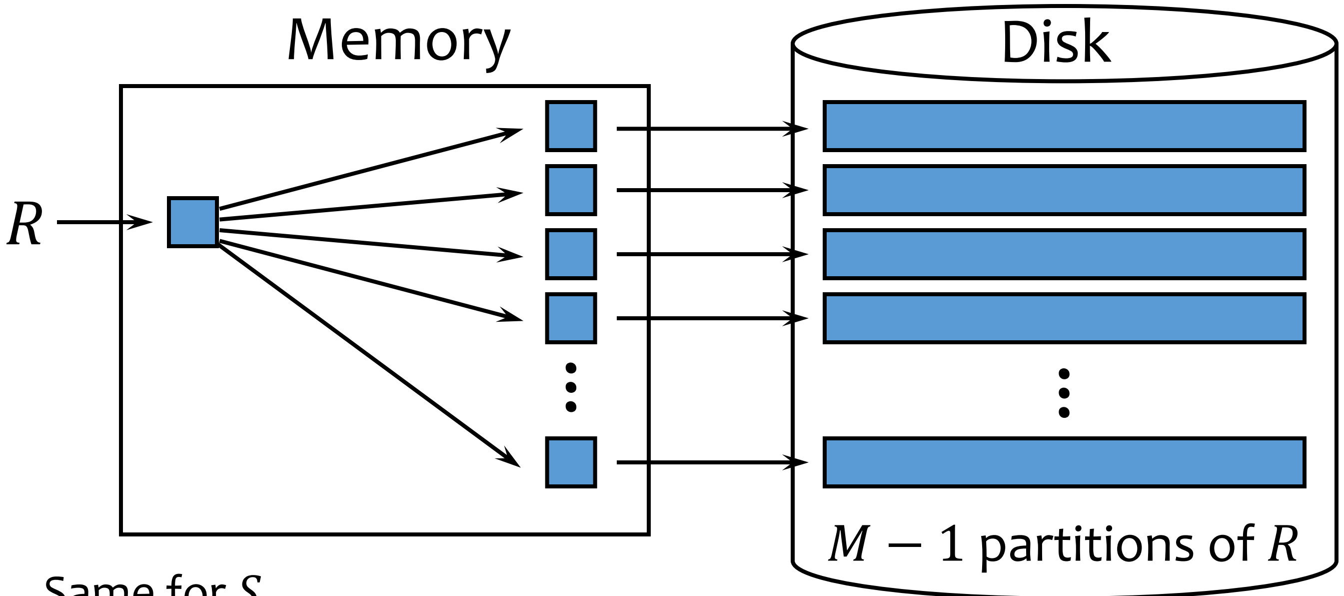


Nested-loop join considers all slots

Hash join considers only those along the diagonal!

Partitioning phase

- Partition R and S according to the same hash function on their join attributes

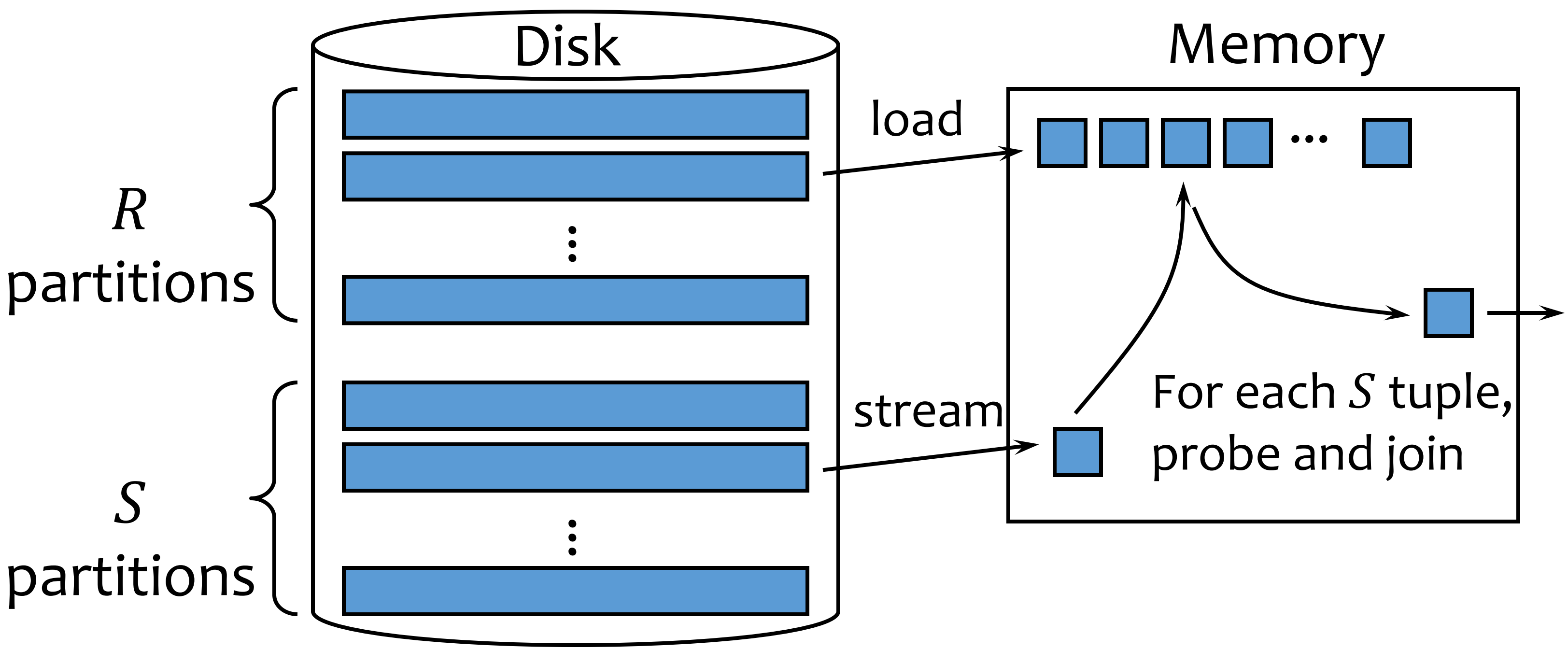


Same for S

Each partition has a size of B(R)/(M-1)

Probing phase

- Read in each partition of R, stream in the corresponding partition of S, join
  - Typically build a hash table for the partition of R
    - Not the same hash function used for partition, of course!



Indexes: Selection using index

- Equality predicate: $\sigma_{A=v}(R)$
  - Use an ISAM, B+-tree, or hash index on $R(A)$
- Range predicate: $\sigma_{A>v}(R)$
  - Use an ordered index (e.g., ISAM or B+-tree) on $R(A)$
  - Hash index is not applicable
- Index-only queries which do not require retrieving actual tuples
  - Example: $\pi_A\left(\sigma_{A>v}(R)\right)$
- Primary index clustered according to search key
  - One lookup leads to all result tuples in their entirety

Index nested-loop join

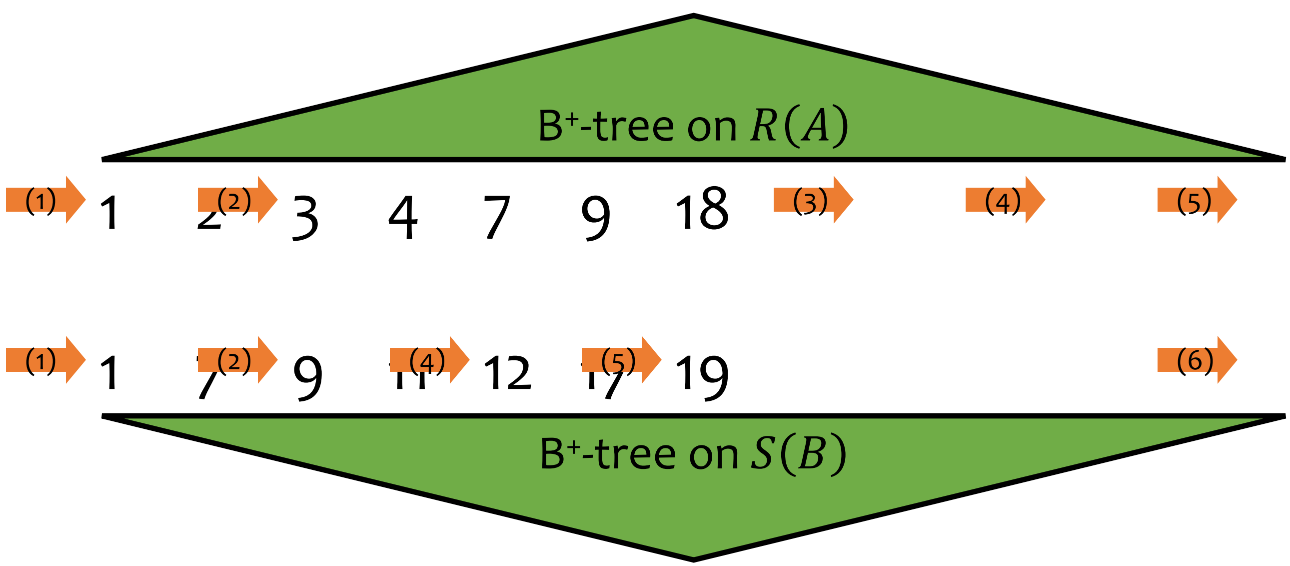
$R \bowtie_{R.A=S.B} S$

- Idea: use a value of $R.A$ to probe the index on $S(B)$
- For each block of R, and for each r in the block:
  Use the index on $S(B)$ to retrieve s with $s.B = r.A$
    Output rs
- I/O's: $B(R) + |R| \cdot$ lookup + fetching cost
  - Typically, the cost of an index lookup is 2-4 I/O's
  - Beats other join methods if $|R|$ is not too big
  - Better pick R to be the smaller relation
- Memory requirement: $M \geq 3$ (blocks)

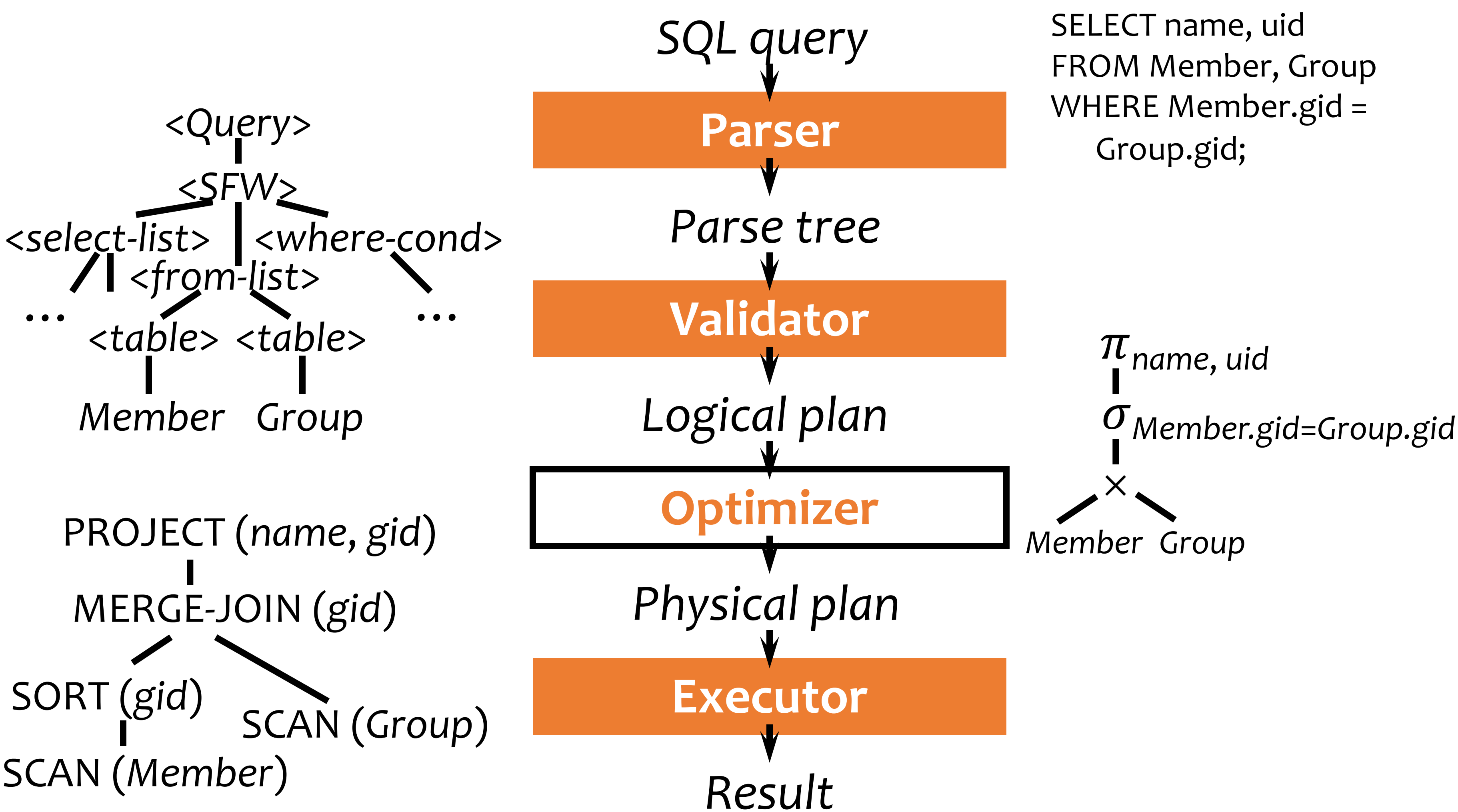
Zig-zag join using ordered indexes

$R \bowtie_{R.A=S.B} S$

- Idea: use the ordering provided by the indexes on $R(A)$ and $S(B)$ to eliminate the sorting step of sort-merge join
- Use the larger key to probe the other index
  - Possibly skipping many keys that don't match

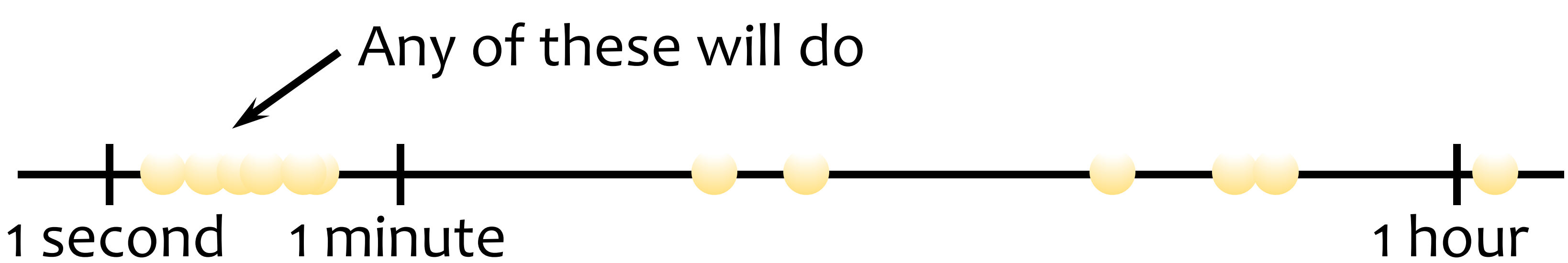


Back to the trip



Query optimization

- Why query optimization?
- Search space
  - What are the possible equivalent logical plans?
  - What are the possible physical plans? (Lecture 16)
- Search strategy
  - Rule-based strategy
  - Cost-based strategy



Algebraic equivalences

- Join reordering: $\times$ and $\bowtie$ are associative and commutative (except column ordering)
- Convert σ_p-$\times$ to/from $\bowtie_p$: $\sigma_p(R \times S) = R \bowtie_p S$
- Merge/split σ's: $\sigma_{p_1}\left(\sigma_{p_2} R\right) = \sigma_{p_1 \land p_2} R$
- Merge/split π's: $\pi_{L_1}\left(\pi_{L_2} R\right) = \pi_{L_1 \land L_2} R$
- Push down σ: (p_R involves only R; p_S involves only S; p and p' involve R and S)

$$\sigma_{p \land p_R \land p_S}\left(R \bowtie_{p'} S\right) = \left(\sigma_{p_R} R\right) \bowtie_{p \land p'} \left(\sigma_{p_S} S\right)$$

- Push down π: $\pi_L\left(\sigma_p R\right) = \pi_L\left(\sigma_p(\pi_{L \cup L'} R)\right)$
  - L' is the set of columns referenced by p

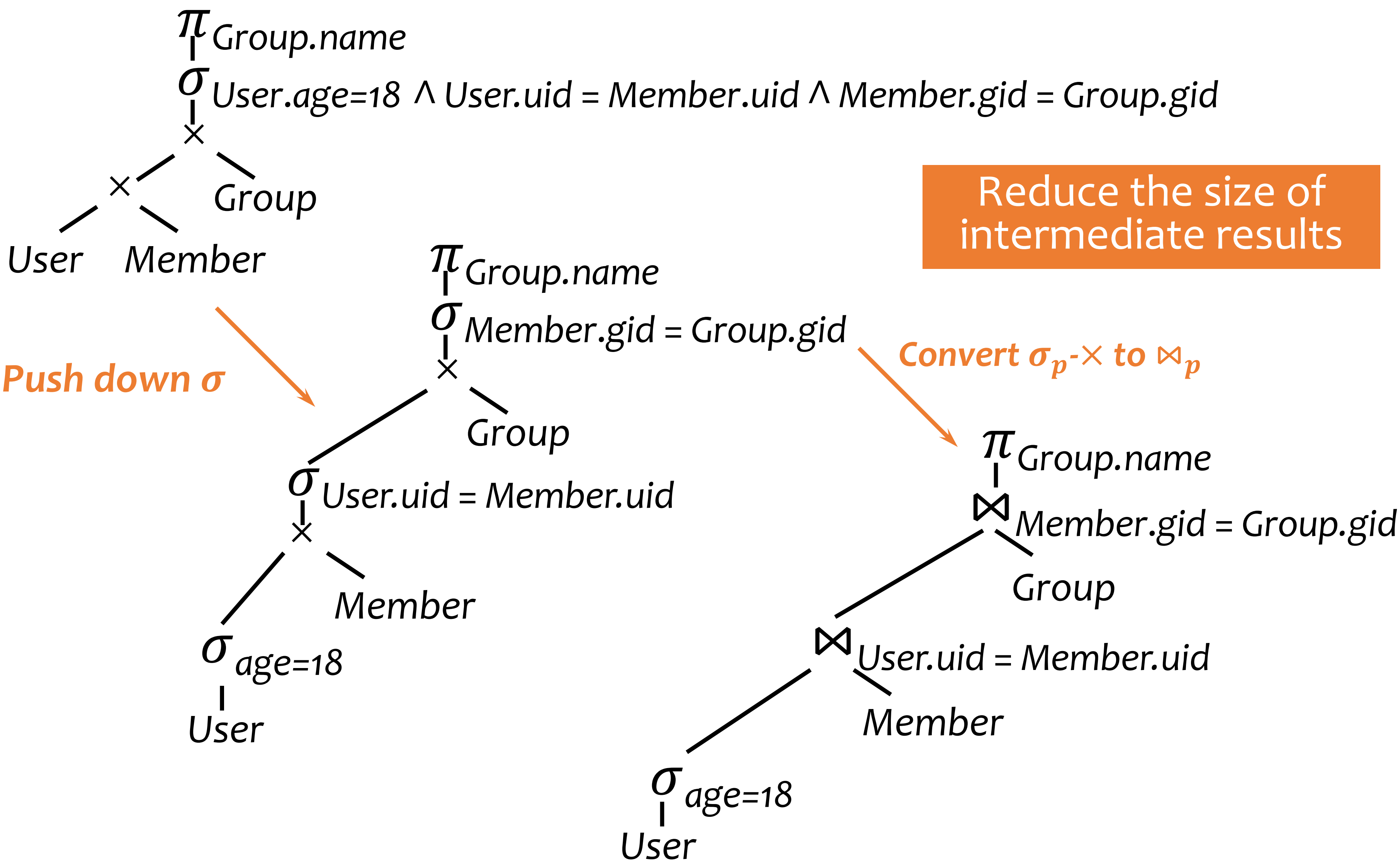
Algebraic equivalences

- Push down π : (L_R is the set of columns referenced by p and L for R; L_S is the set of columns referenced by p and L for S)

$$\pi_L(R \bowtie_p S) = \pi_L\left((\pi_{L_R} R) \bowtie_p (\pi_{L_S} S)\right)$$

- Push down $-$:
  - Suppose R and T have the same schema:
    $$(R \bowtie S) - (T \bowtie S) = (R - T) \bowtie S$$
  - Suppose S and W also have the same schema
    $$(R \bowtie S) - (T \bowtie W)$$
    $$= \left((\mathrm{R} - \mathrm{T}) \bowtie \mathrm{S}\right) \cup \left(\mathrm{R} \bowtie (\mathrm{S} - \mathrm{W})\right)$$

Rule-based query optimization



Selections with equality predicates

Consider $\sigma_{A=v} R$

- DBMSs typically store the following in the catalog
  - Size of R: $|R|$
  - Number of distinct A values in R: $|\pi_A R|$
- Assumption of uniformity: A-values are uniformly distributed in tuples from R
- $|\sigma_{A=v} R| \approx {}^{|R|}/_{|\pi_A R|}$
  - Selectivity factor of $(A = v)$ is ${}^{1}/_{|\pi_A R|}$
  - Selectivity: the probability that any row will satisfy a predicate

Conjunctive predicates

Consider $\sigma_{A=u \land B=v} R$

- Assumption of selection independence: $(A = u)$ and $(B = v)$ independently select tuple in R
  - Counterexample: major and advisor, or A is the key
- $|\sigma_{A=u \land B=v} R| \approx {}^{|R|}/_{|\pi_A R| \cdot |\pi_B R|}$
  - Selectivity factor of $(A = u)$ is ${}^{1}/_{|\pi_A R|}$
  - Selectivity factor of $(B = v)$ is ${}^{1}/_{|\pi_B R|}$
  - Selectivity factor of $(A = u) \land (B = v)$ is ${}^{1}/_{|\pi_A R| \cdot |\pi_B R|}$
  - Reduce total size by all selectivity factors

Negated and disjunctive predicates

Consider $\sigma_{A \neq v} R$

- $|\sigma_{A \neq v} R\,| \approx |R| \cdot \left(1 - {}^{1}/_{|\pi_A R|}\right)$
  - Selectivity factor of $\neg p$ is $(1 -$ selectivity factor of $p)$

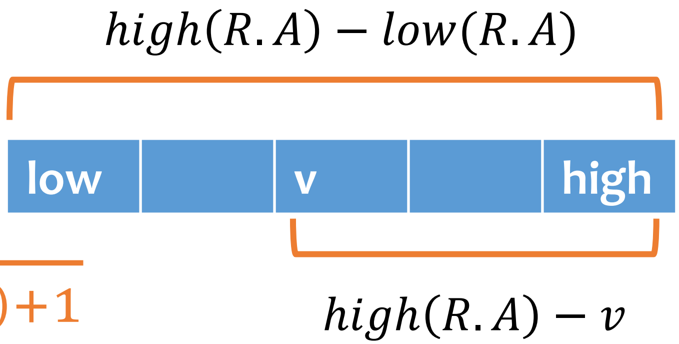
Consider $\sigma_{A=u \,\vee\, B=v} R$

- $|\sigma_{A=u \,\vee\, B=v} R| \approx |R| \cdot \left({}^{1}/_{|\pi_A R|} + {}^{1}/_{|\pi_B R|}\right)$?
  - Tuples satisfying $(A = u)$ and $(B = v)$ are counted twice!
- $|\sigma_{A=u \,\vee\, B=v} R| \approx |R| \cdot \left({}^{1}/_{|\pi_A R|} + {}^{1}/_{|\pi_B R|} - {}^{1}/_{|\pi_A R||\pi_B R|}\right)$
  - Inclusion-exclusion principle

Range predicates

Consider $\sigma_{A>v}R$

- DBMSs typically store the following in the catalog
  - Largest *R.A* value: $\mathrm{high}(R.A)$
  - Smallest *R.A* value: $\mathrm{low}(R.A)$



- $|\sigma_{A>v}R| \approx |R| \cdot \frac{\mathrm{high}(R.A)-v}{\mathrm{high}(R.A)-\mathrm{low}(R.A)+1}$
  - Selectivity factor is $\frac{\mathrm{high}(R.A)-v}{\mathrm{high}(R.A)-\mathrm{low}(R.A)}$

Two-way natural join

- $Q = R(A, B) \bowtie S(A, C)$
- Assumption of containment of value sets: every tuple in the "smaller" relation (one with fewer distinct values for the join attribute) joins with some tuple in the other relation
  - That is, if $|\pi_A R| \leq |\pi_A S|$ then $\pi_A R \subseteq \pi_A S$
  - Certainly not true in general
  - But holds many practical cases
- $|Q| \approx \frac{|R| \cdot |S|}{\max(|\pi_A R|, |\pi_A S|)}$
  - Selectivity factor of $R.A = S.A$ is ${}^{1}/_{\max(|\pi_A R|, |\pi_A S|)}$

Multiway natural join

- Q: $R(A,B) \bowtie S(B,C) \bowtie T(C,D)$
- What is the number of distinct C values in the join of R and S?
- Assumption of preservation of value sets
  - A non-join attribute does not lose values from its set of possible values
  - That is, if C is in S but not R, then $\pi_C(R \bowtie S) = \pi_C S$
  - Certainly not true in general
  - But holds many practical cases

Multiway natural join (cont'd)

- Q: $R(A,B) \bowtie S(B,C) \bowtie T(C,D)$
- Reduce the total size by the selectivity factor of each join predicate
  - $R.B = S.B$: ${}^{1}/_{\max(|\pi_B R|, |\pi_B S|)}$
  - $|R \bowtie S| = {}^{|R| \cdot |S|}/_{\max(|\pi_B R|, |\pi_B S|)}$
  - $(R \bowtie S).C = T.C$: ${}^{1}/_{\max(|\pi_C (R \bowtie S)|, |\pi_C T|)} = {}^{1}/_{\max(|\pi_C S|, |\pi_C T|)}$
  - $|Q| \approx \dfrac{|R| \cdot |S| \cdot |T|}{\max(|\pi_B R|, |\pi_B S|) \cdot \max(|\pi_C S|, |\pi_C T|)}$

Summary of Cardinality Estimation

- Lots of assumptions and very rough estimation
  - An accurate estimator is not needed
  - Maybe okay if we overestimate or underestimate, since it may not change the query plan selection

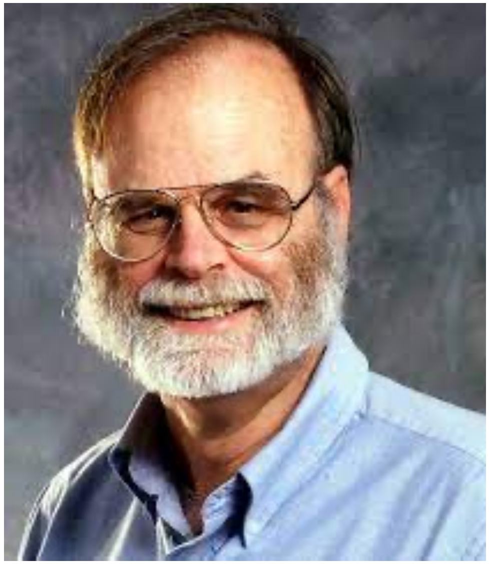
- Pay attention to the assumptions!

Transaction

Transactions

```
-- Begins implicitly
SELECT … ;
UPDATE … ;
ROLLBACK | COMMIT
```

- A transaction is a sequence of database operations (read or write)
- ACID properties of transactions (TXs)
  - Atomicity: TXs are either completely done or not done at all (next lecture)
  - Consistency: TXs should leave the database in a consistent state
  - Isolation: TXs must behave as if they execute in isolation (this-next lecture)
  - Durability: Effects of committed TXs are resilient against failures (next lecture)



Jim Gray, Turing Award 1998, who coined this term (as well as data cube and many other things)

Different Isolation Levels

Stronger Consistency

Higher Overheads

Less Concurrency

[Isolation Levels in SQL Standard](#)
Read Uncommitted
Read Committed
Repeatable Read
Serializable

Weaker Consistency

Lower Overheads

More Concurrency

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRANSACTION;
SELECT * FROM Order;
...
COMMIT TRANSACTION
```

READ UNCOMMITTED

- Can read "dirty" data
  - A data item is dirty if it is written by an uncommitted transaction
- Problem: What if the transaction that wrote the dirty data eventually aborts?
- Example: wrong average
  -

-- T1:	-- T2:
UPDATE User	
SET pop = 0.99	
WHERE uid = 142;	SELECT AVG(pop)
	FROM User;
ROLLBACK;	
	COMMIT;

READ COMMITTED

- No dirty reads, but non-repeatable reads possible
  - Reading the same data item twice can produce different results
- Example: different averages
  - -- T1:

```
-- T1:                          -- T2:
                                SELECT AVG(pop)
                                FROM User;
UPDATE User
SET pop = 0.99
WHERE uid = 142;
COMMIT;
                                SELECT AVG(pop)
                                FROM User;
                                COMMIT;
```

REPEATABLE READ

- Reads are repeatable, but may see phantoms
  - Reading the same data item twice still see the same value
  - But some new data item may appear
- Example: different average (still!)
  - -- T1: -- T2:

```
-- T2:
SELECT AVG(pop)
FROM User;
```

```
-- T1:
INSERT INTO User
VALUES(789, 'Nelson',10, 0.1);
COMMIT;
```

```
-- T2:
SELECT AVG(pop)
FROM User;
COMMIT;
```

SERIALIZABLE

- All three anomalies can be avoided:
  - No dirty reads
  - No non-repeatable reads
  - No phantoms

- For any two transactions T1 and T2:
  - T1 followed by T2 or T2 followed by T1

Execution histories of Transactions

- A transaction is an ordered sequence of read or write operations on the database, followed by abort or commit.
  - Database is *a set* of independent data items x, y, z etc.
  - T = {read(x), write(y), read(z), write(z), write(x), commit}

- An execution history over a set of transactions $T_1 \ldots T_n$ is an interleaving of the operations of $T_1 \ldots T_n$ in which the operation ordering imposed by each transaction is preserved.
  - Transactions interact with each other only via reads and writes of the same date item

Examples for valid execution history

- $T_1 = \{w_1[x], w_1[y], c_1\}, T_2 = \{ r_2[x], r_2[y], c_2 \}$

T_1	T_2
w1(x)	
	r2(x)
w1(y)	
	r2(y)
c1	
	c2

H_a

T_1	T_2
w1(x)	
w1(y)	
c1	
	r2(x)
	r2(y)
	c2

H_b

T_1	T_2
w1(x)	
	r2(x)
	r2(y)
w1(y)	
c1	
	c2

H_c

T_1	T_2
	r2(x)
	r2(y)
	c2
w1(x)	
w1(y)	
c1	

H_d

Serial execution histories

no interleaving operations from different transactions

- $T_1 = \{w_1[x], w_1[y], c_1\}, T_2 = \{ r_2[x], r_2[y], c_2 \}$

T_1	T_2
w1(x)	
	r2(x)
w1(y)	
	r2(y)
c1	
	c2

H_a

T_1	T_2
w1(x)	
w1(y)	
c1	
	r2(x)
	r2(y)
	c2

H_b ✓

T_1	T_2
w1(x)	
	r2(x)
	r2(y)
w1(y)	
c1	
	c2

H_c

T_1	T_2
	r2(x)
	r2(y)
	c2
w1(x)	
w1(y)	
c1	

H_d ✓

Equivalence of execution histories

- Two operations conflict if
  - they belong to different transactions,
  - they operate on the same data item, and
  - at least one of the operations is write
    - two types of conflicts: read-write and write-write

- Two execution histories are (conflict) equivalent if
  - they are over the same set of transactions
  - the ordering of each pair of conflicting operations is the same in each history

Serializable

- A history H is said to be (conflict) serializable if there is some serial history H' (conflict) equivalent to H.

T_1	T_2
w1(x)	
	r2(x)
w1(y)	
	r2(y)
c1	
c2	

T_1	T_2
w1(x)	
w1(y)	
c1	
	r2(x)
	r2(y)
	c2

$H_a = H_b$

T_1	T_2
w1(x)	
	r2(x)
	r2(y)
w1(y)	
c1	
c2	

?

Serializable

- Serialization graph (V, E) for history H:
  - $V = \{T\colon\ T$ is a committed transaction in $H\}$
  - $E = \{T_i \rightarrow T_j\colon \exists\ o_i \in T_i$ and $o_j \in T_j$ conflict; and $o_i < o_j\}$

Two operations conflict if
- they belong to different transactions;
- they operate on the same data item;
- at least one of the operations is write

- A history is serializable if and only if its serialization graph is acyclic (i.e., no cycles)

Example

- Example: $H_a = w_1[x]r_2[x]w_1[y]r_2[y]c_1c_2$

T_1	T_2
w1(x)	
	r2(x)
w1(y)	
	r2(y)
c1	
	c2

H_a

$w_1[x]$ and $r_2[x]$ conflict, and $w_1[x] < r_2[x]$
$w_1[y]$ and $r_2[y]$ conflict, and $w_1[y] < r_2[y]$

$T_1 \rightarrow T_2$

no cycles,
so serializable

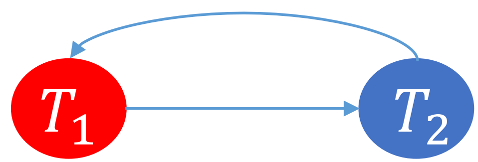
Example

- Example: $H_c = w_1[x] r_2[x] r_2[y] w_1[y] \mathrm{c}_1 \mathrm{c}_2$

T_1	T_2
w1(x)	
	r2(x)
	r2(y)
w1(y)	
c1	
	c2

H_c

$w_1[x]$ and $r_2[x]$ conflict, and $w_1[x] < r_2[x]$;
$w_1[y]$ and $r_2[y]$ conflict, and $r_2[y] < w_1[y]$



Not serializable

Locking

(Pessimistic) Assume that conflicts will happen and take preventive action

- If a transaction wants to read x , it must first request a shared lock (S mode) on x
- If a transaction wants to modify x, it must first request an exclusive lock (X mode) on x
- Allow one exclusive lock, or multiple shared locks

Mode of the lock requested

Mode of lock currently held by other transactions

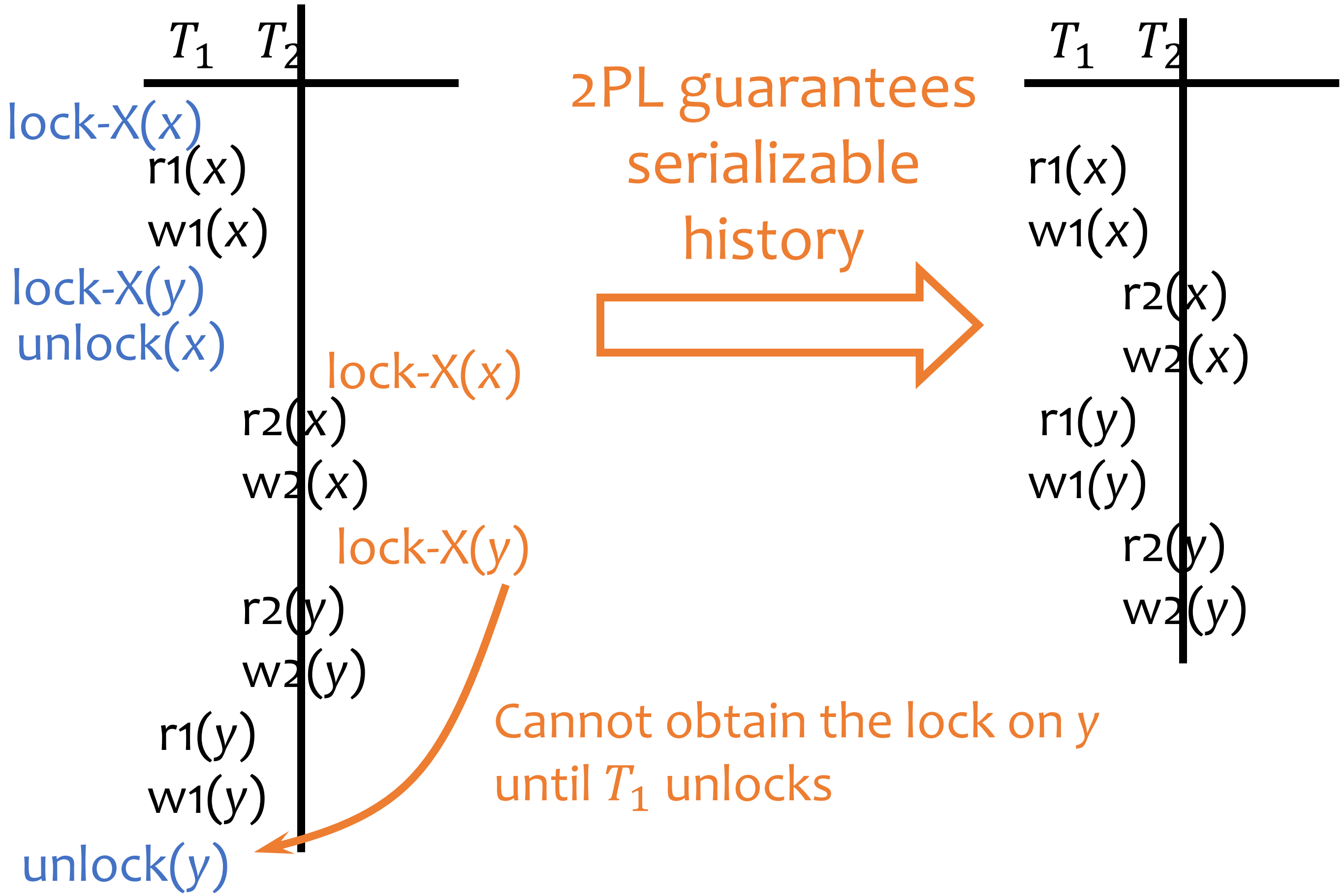
	lockS	lockX
lockS	Yes	No
lockX	No	No

Grant the lock?

Compatibility matrix

Two-phase locking (2PL)

- All lock requests precede all unlock requests
  - Phase 1: obtain locks; Phase 2: release locks



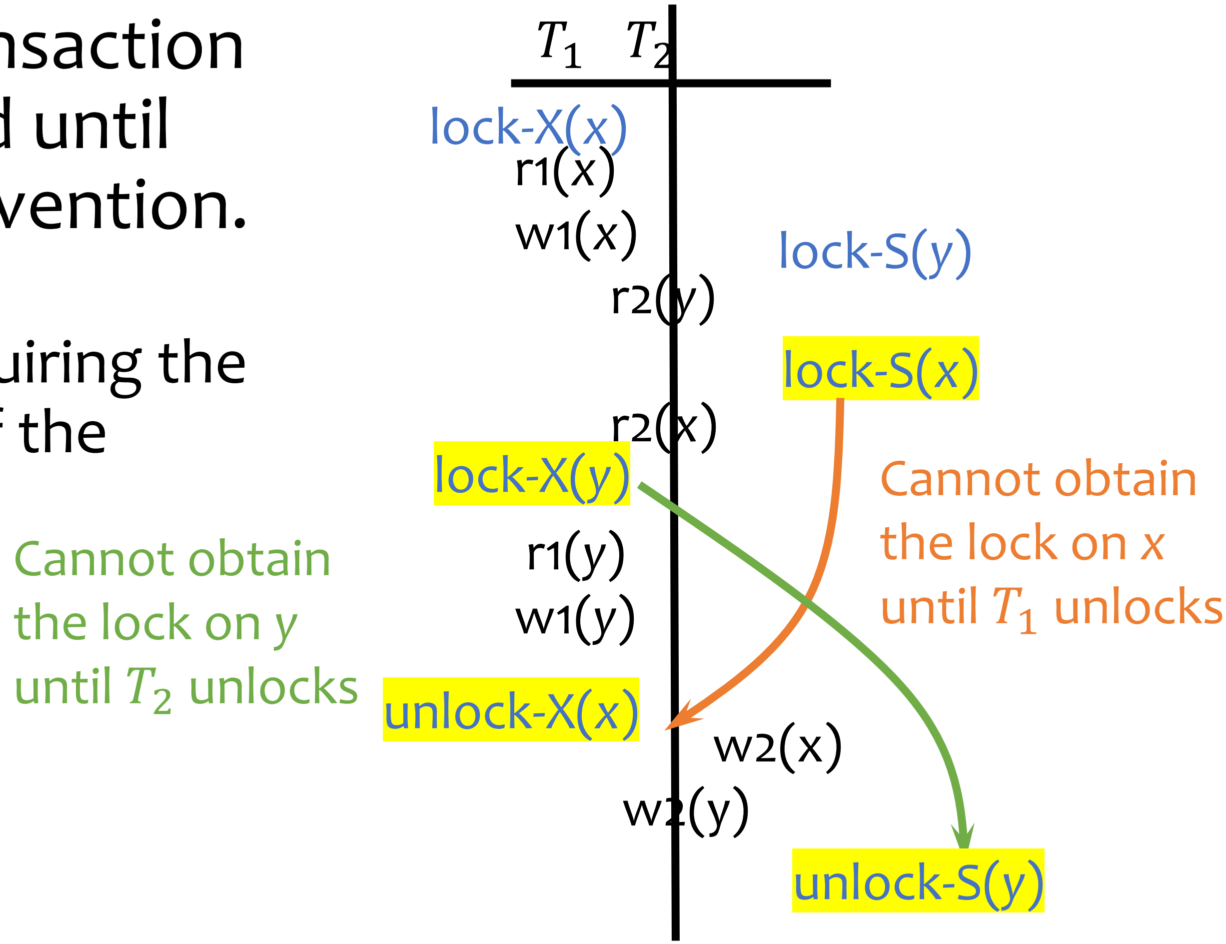
Remaining problems of 2PL

T_1	T_2
lock-X(x)	
r1(x)	
w1(x)	
lock-X(y)	
unlock(x)	
	lock-X(x)
	r2(x)
	w2(x)
r1(y)	
w1(y)	
unlock(y)	
	lock-X(y)
	r2(y)
	w2(y)
	unlock(x)
	unlock(y)
	commit
abort	

- T_2 has read uncommitted data written by T_1
- If T_1 aborts, then T_2 must abort as well
- Cascading aborts possible if other transactions have read data written by T_2
- Even worse, schedule is not recoverable if T_2 commits before T_1

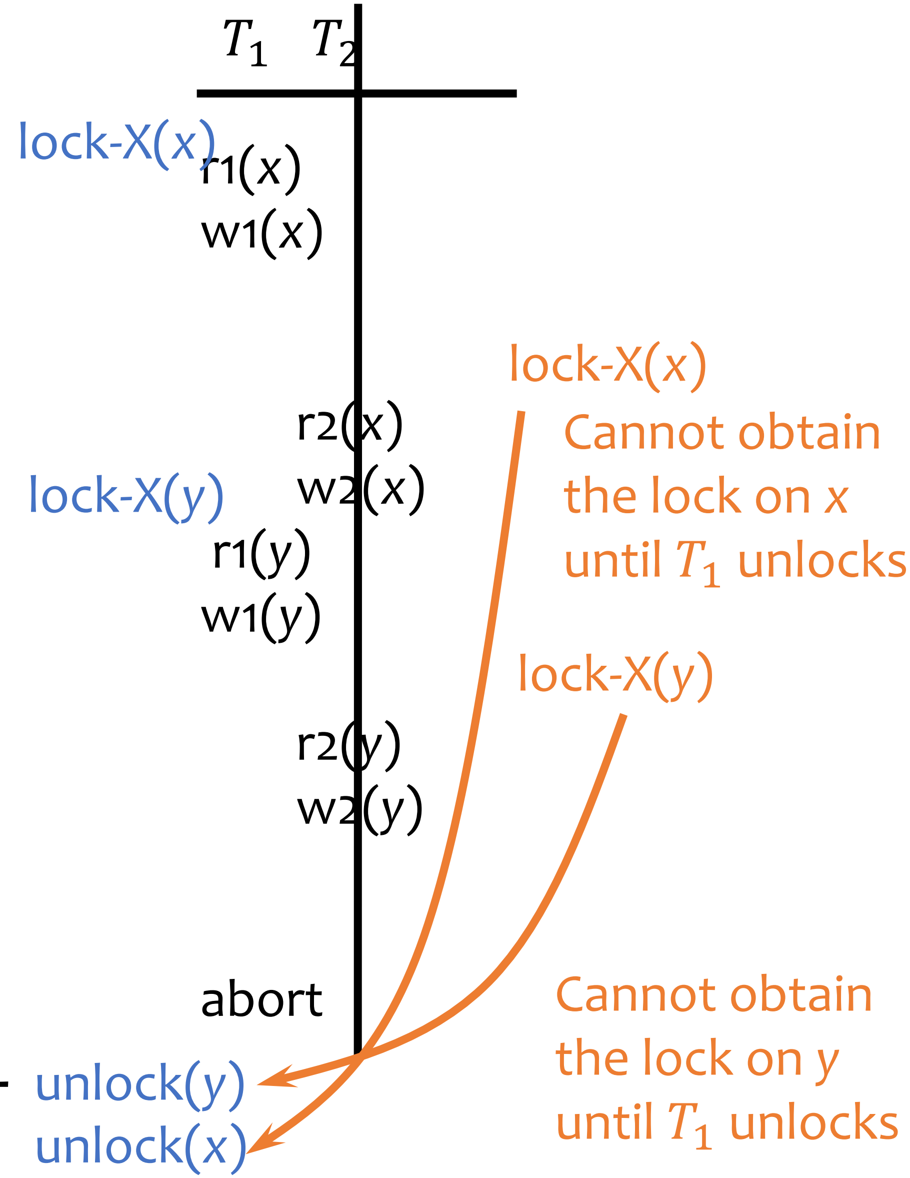
Remaining problems of 2PL

- Deadlock: A transaction remains blocked until there is an intervention.
  - 2PL may cause deadlocks, requiring the abort of one of the transactions



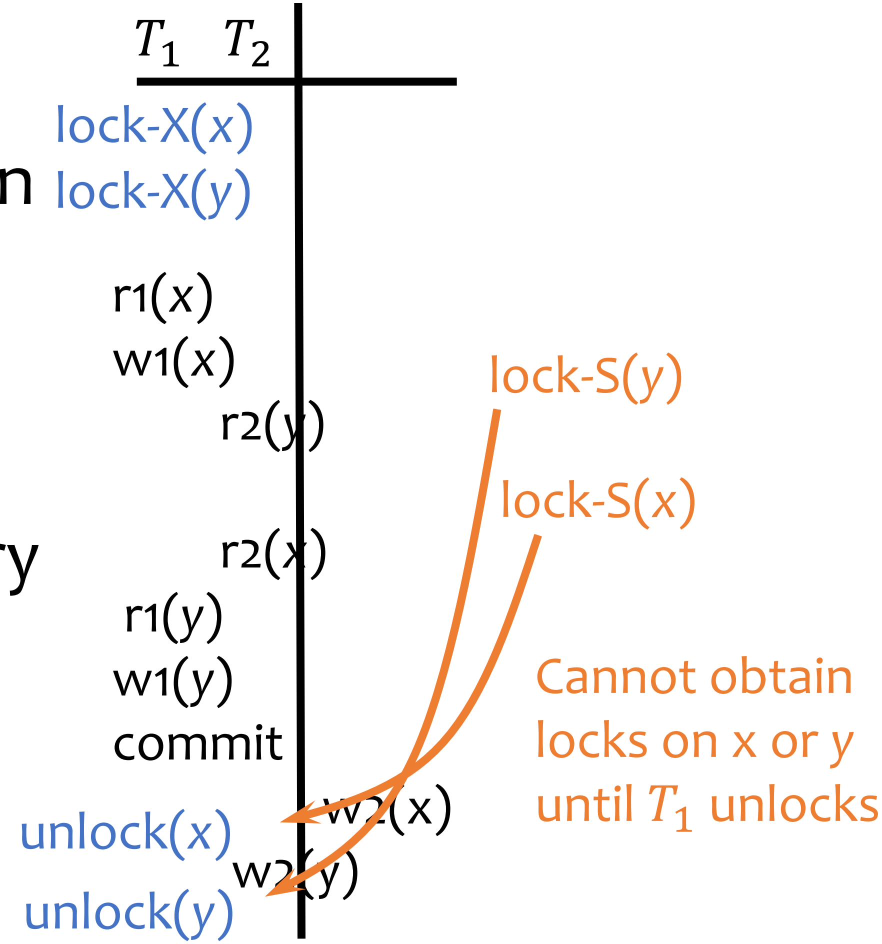
Strict 2PL

- Only release X-locks when commit/abort
  - A write will block all other reads until the write commits or aborts
- Used in many practical DBMSs
  - No cascading aborts
  - But it can still lead to deadlocks! (see slide 14)
- Less concurrency than 2PL



Conservative 2PL

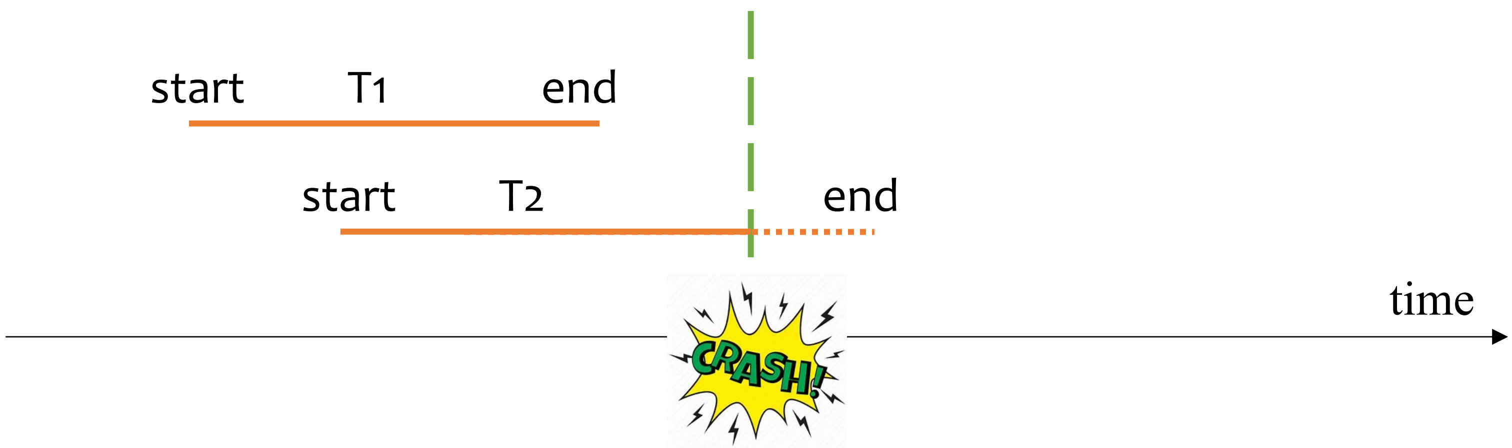
- Only acquire locks at the beginning of the transaction and release X-locks when commit/abort
- Not practical due to the very limited concurrency
  - No cascading aborts
  - No deadlocks



Failures

- System crashes right after a transaction *T1* commits; but not all effects of *T1* were written to disk
  - How do we complete/redo *T1* (durability)?

- System crashes in the middle of a transaction *T2*; partial effects of *T2* were written to disk
  - How do we undo *T2* (atomicity)?



Log

- When a transaction T starts: $\langle T, \text{start}\rangle$
- Record values before and after each modification of data item X: $\langle T, X, \textit{old\_value\_of\_X}, \textit{new\_value\_of\_X}\rangle$
- When a transaction T commits: $\langle T, \text{commit}\rangle$
- When a transaction T aborts: $\langle T, \text{abort}\rangle$

Write-ahead logging (WAL): Before X is modified on disk, the log record pertaining to X must be flushed

Log

$\langle\ T_1$, start $\rangle$

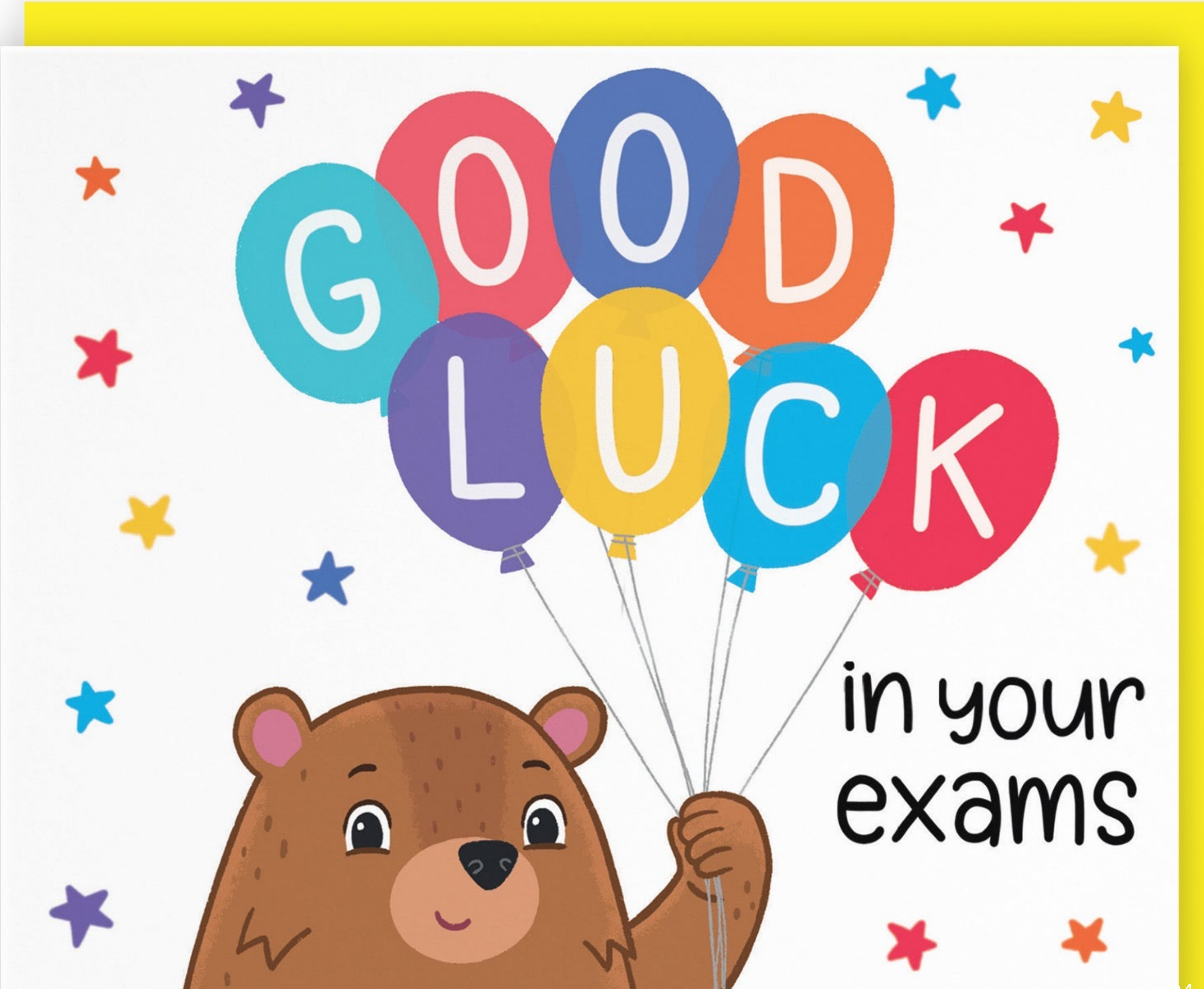
$\langle\ T_1$, A, 800, 700 $\rangle$

$\langle\ T_1$, B, 400, 500 $\rangle$

$\langle\ T_1$, commit $\rangle$

Undo/redo logging - repeat history!

- U: track the set of active transactions at crash
- Redo phase: scan forward to the end of the log
  - For a log record ⟨ T, start ⟩, add T to U
  - For a log record ⟨ T, X, *old*, *new* ⟩, issue write(X, *new*)
  - For a log record ⟨ T, commit | abort ⟩, remove T from U
    - *If abort, undo changes of T* i.e., for a log record ⟨ T, X, *old*, *new* ⟩, issue write(X, *old*)
- Undo phase: scan backward to the start of the log
  - Undo the effects of transactions in U
  - For a log record ⟨ T, X, *old*, *new* ⟩ where T is in U, issue write(X, *old*), and log this operation too, i.e., add ⟨ T, X, *old* ⟩
  - Log ⟨T, abort ⟩ when all effects of T have been undone


GOOD
LUCK
in your
exams