CS 186 Guerrilla Section 2

B+ Trees, Buffer Management, Relational Algebra, Sorting/Hashing

Agenda

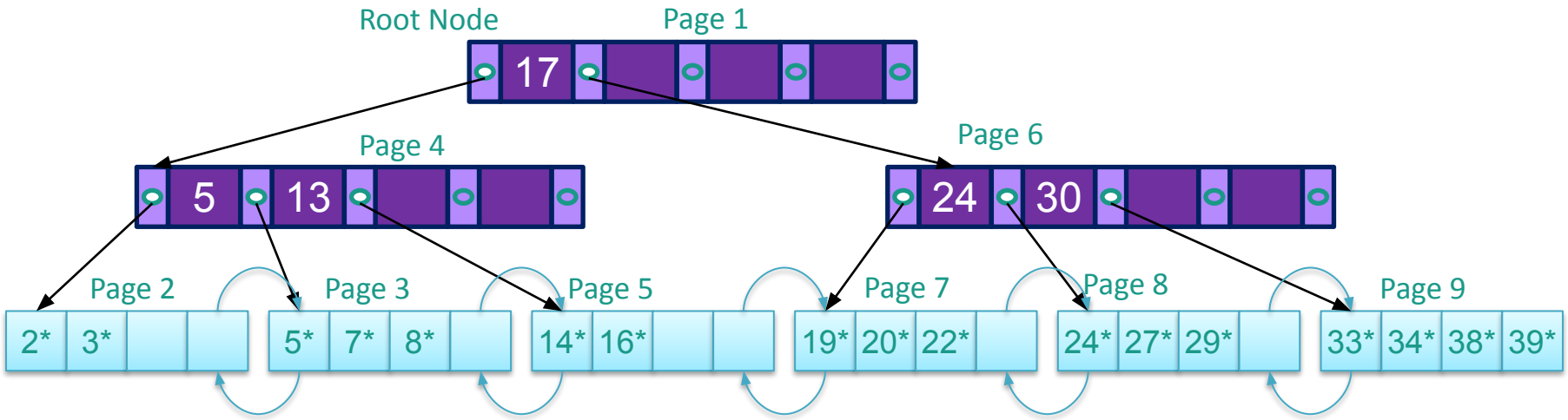
I. 5:10-5:30: Mini-lecture
II. 5:30-6:30: Worksheet
III. 6:30-7:00: Solutions

Worksheet and slides can be found at @171 on Piazza

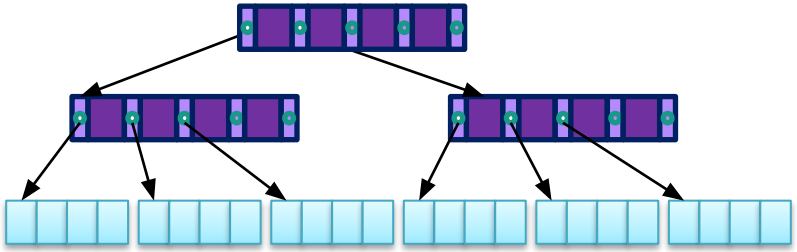
Feedback Form: https://tinyurl.com/CS186ExamPrep2FA20

B+ Trees

B+ Trees

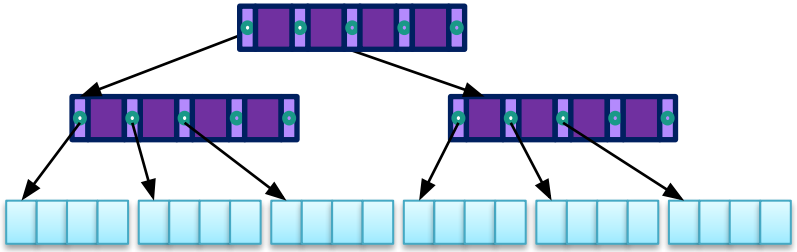

Root Node
Page 1
17
Page 4
5
13
Page 6
24
30
Page 2
2*
3*
Page 3
5*
7*
8*
Page 5
14*
16*
Page 7
19*
20*
22*
Page 8
24*
27*
29*
Page 9
33*
34*
38*
39*

B+ Trees



- Leaf nodes (nodes at the bottom of the tree) contain data entries (e.g. <key, recordId>)
- Inner (or interior) nodes (nodes not at the bottom) are **solely** for lookup
  - No data entries are stored in these!

B+ Trees



- We use *d* to denote the order of the tree, each node contains up to *2d* values
  - Inner nodes have a maximum fanout of *2d+1*
- Key Invariant: for a value *x* in the inner node, the corresponding subtree only has values ≥x
- Height: the number of pointers to take from root to leaf
  - Alternatively, the number of inner node levels in the tree

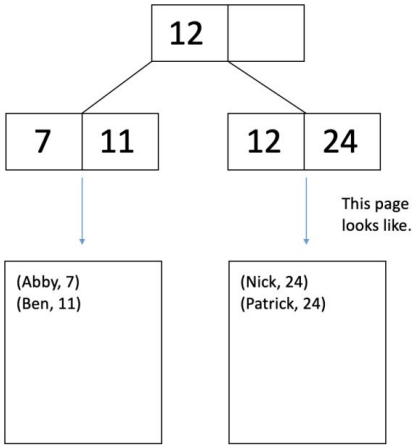
Index types

How is data stored in the index?

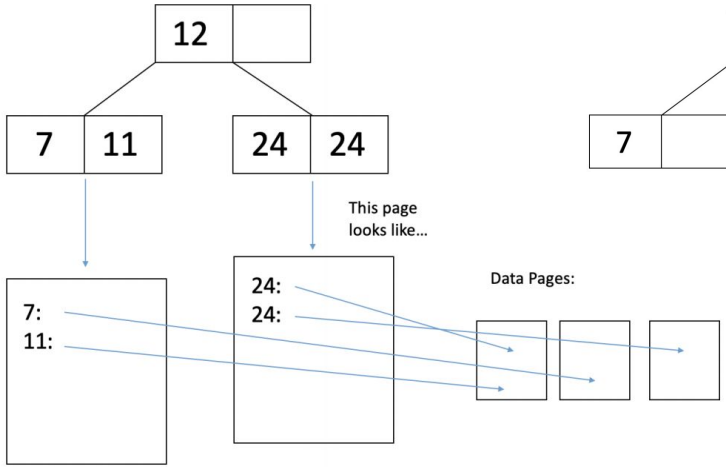
- Alternative 1: store actual records in the index (leaf nodes)
  - We can't do this efficiently if we have multiple indices on the table (need to keep multiple copies)
- Alternative 2: store <key, recordId> in the index
- Alternative 3: store <key, list of matching record ids>
  - More compact than Alternative 2 if we have a lot of duplicates

How is data stored in the index?

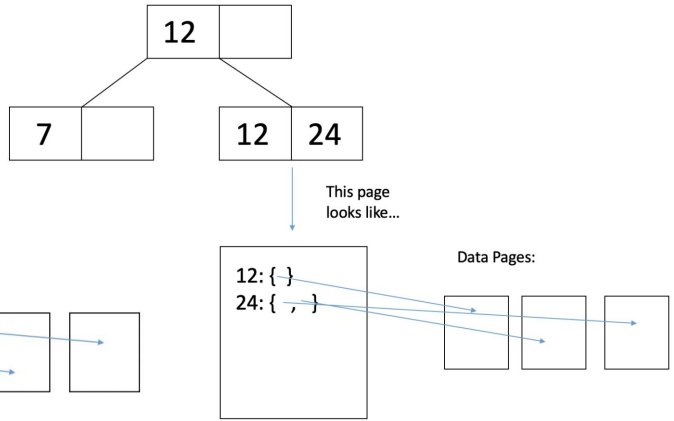
Alternative 1



Alternative 2



Alternative 3

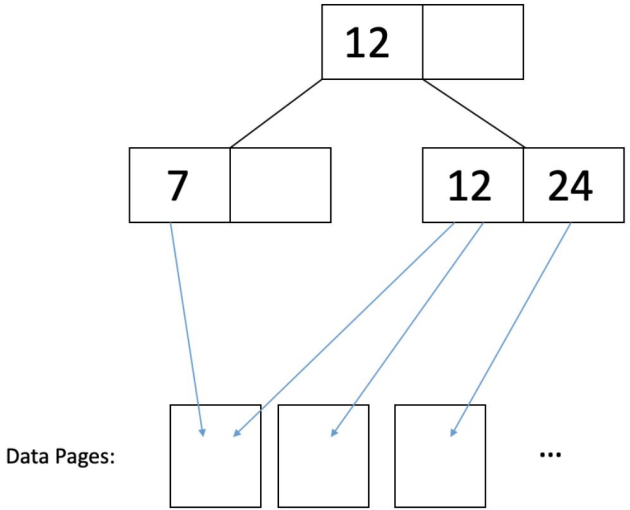


Clustering

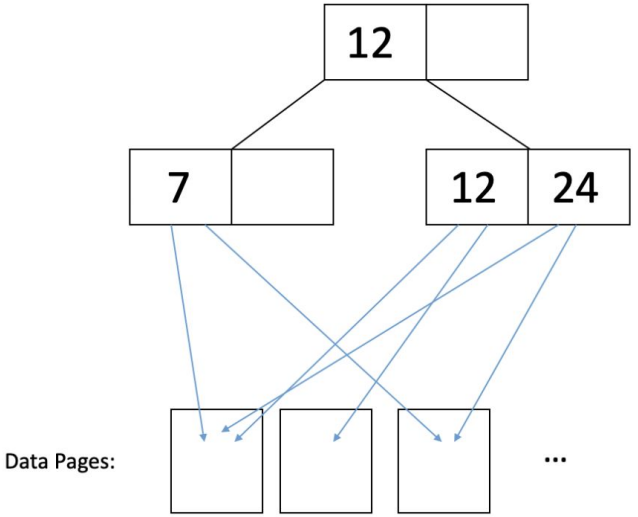
- Clustered index: the actual data is (roughly) sorted in order of the search key
  - Can’t have two clustered indices on different columns without duplicating the data
- Unclustered index: the actual data is *not* in order of the search key
  - Much less efficient

Clustering

Clustered index



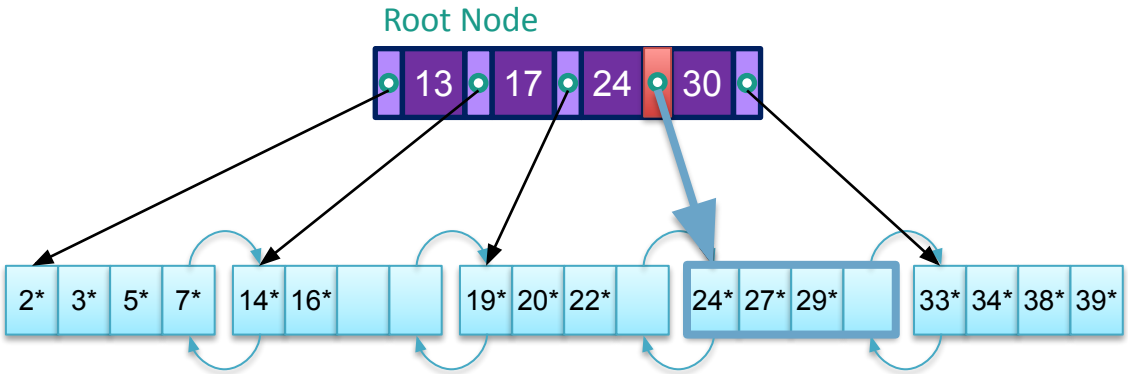
Unclustered index



Inserting into B+ trees

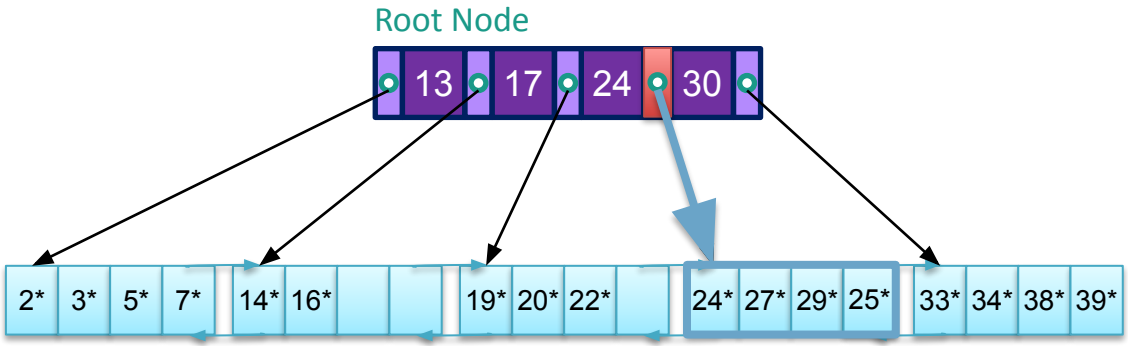
B+ trees: insert 25

- Find the correct leaf node



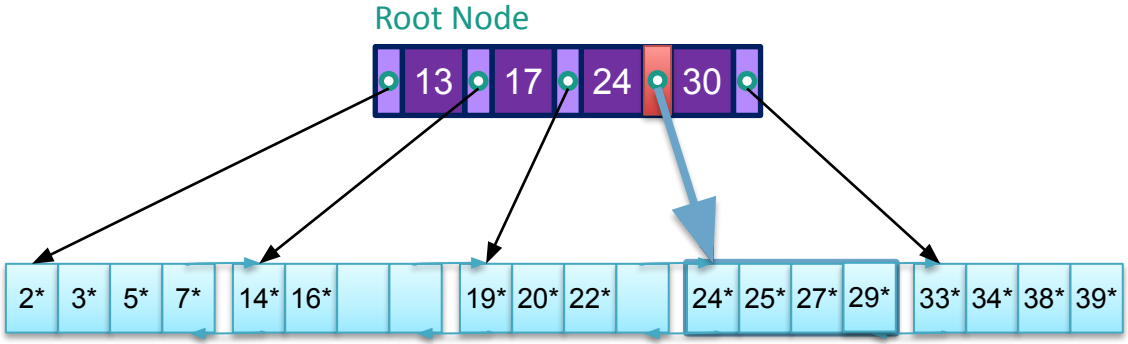
B+ trees: insert 25

- Find the correct leaf node
- If there is room in leaf, just add the entry



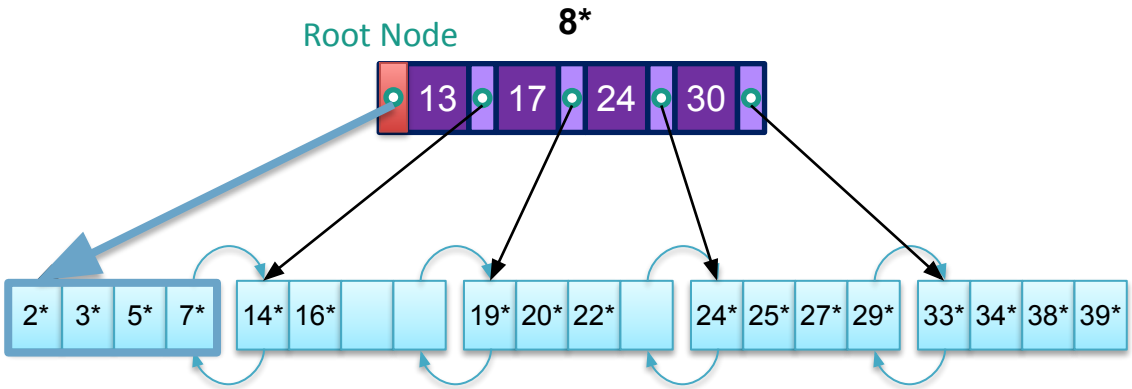
B+ trees: insert 25

- Find the correct leaf node
- If there is room in leaf, just add the entry
  - ..and keep the leaf node sorted



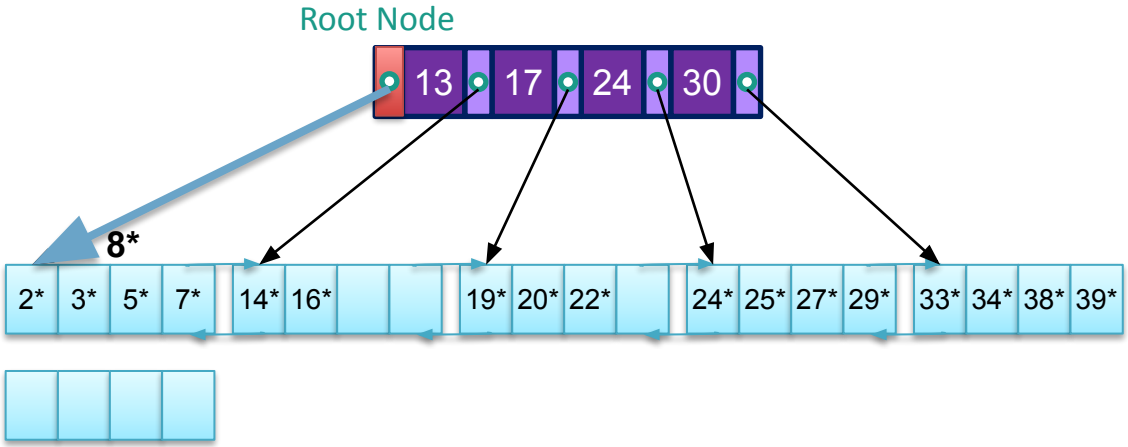
B+ trees: insert 8

- Find the correct leaf node



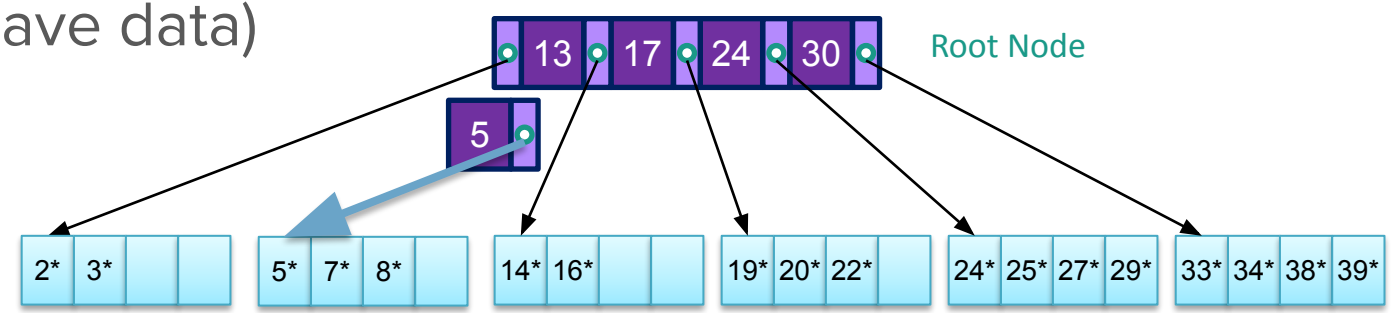
B+ trees: insert 8

- Find the correct leaf node
  - Split leaf if not enough room: into two leaves with d and d+1 entries



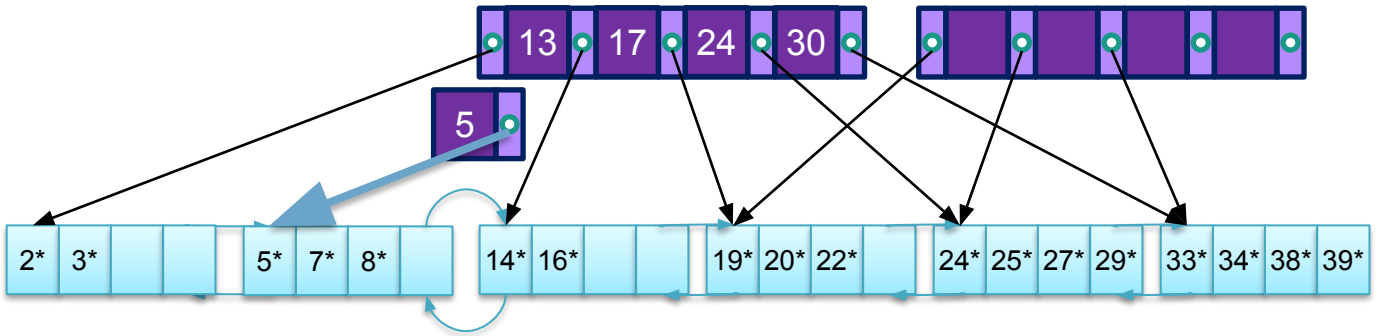
B+ trees: insert 8

- Find the correct leaf node
  - Split leaf if not enough room: into d and d+1 entries
  - **Copy** up the middle key to inner node (since leaf nodes have data)



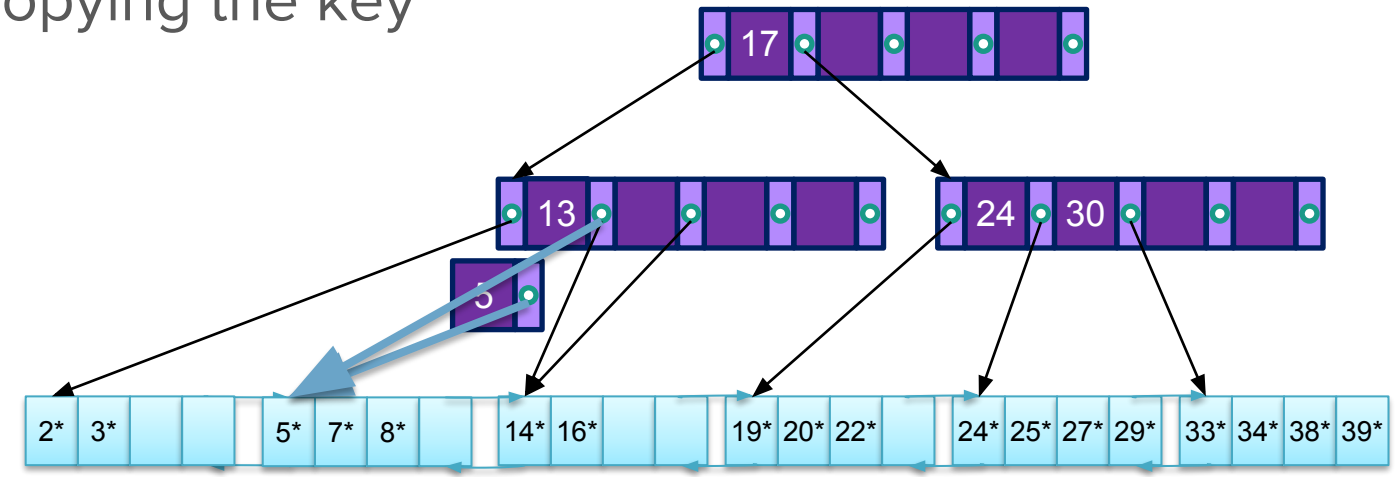
B+ trees: insert 8

- If inner node is full, split the inner node into two and **push** the middle key up
  - Inner nodes don't contain any data, so there's no point copying the key



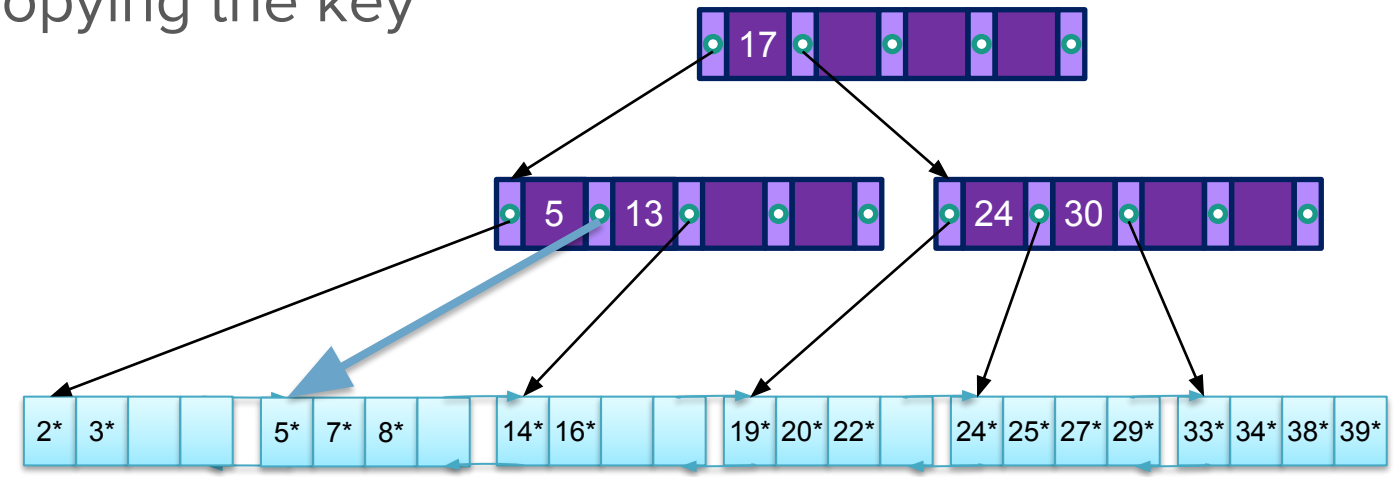
B+ trees: insert 8

- If inner node is full, split the inner node into two and **push** the middle key up
  - Inner nodes don't contain any data, so there's no point copying the key



B+ trees: insert 8

- If inner node is full, split the inner node into two and **push** the middle key up
  - Inner nodes don't contain any data, so there's no point copying the key



Runtime

	Heap File	Sorted File	Clustered Index
Scan all records	B*D	B*D	3/2 * B * D
Equality Search	0.5*B*D	$(\log_2 B)*D$	$(\log_F(BR/E)+2)*D$
Range Search	B*D	$((\log_2 B)+pages))*D$	$(\log_F(BR/E)+3*pages)*D$
Insert	2*D	$((\log_2 B) + B)*D$	$(\log_F(BR/E)+4)*D$
Delete	(0.5*B+1)*D	$((\log_2 B) + B)*D$	$(\log_F(BR/E)+4)*D$

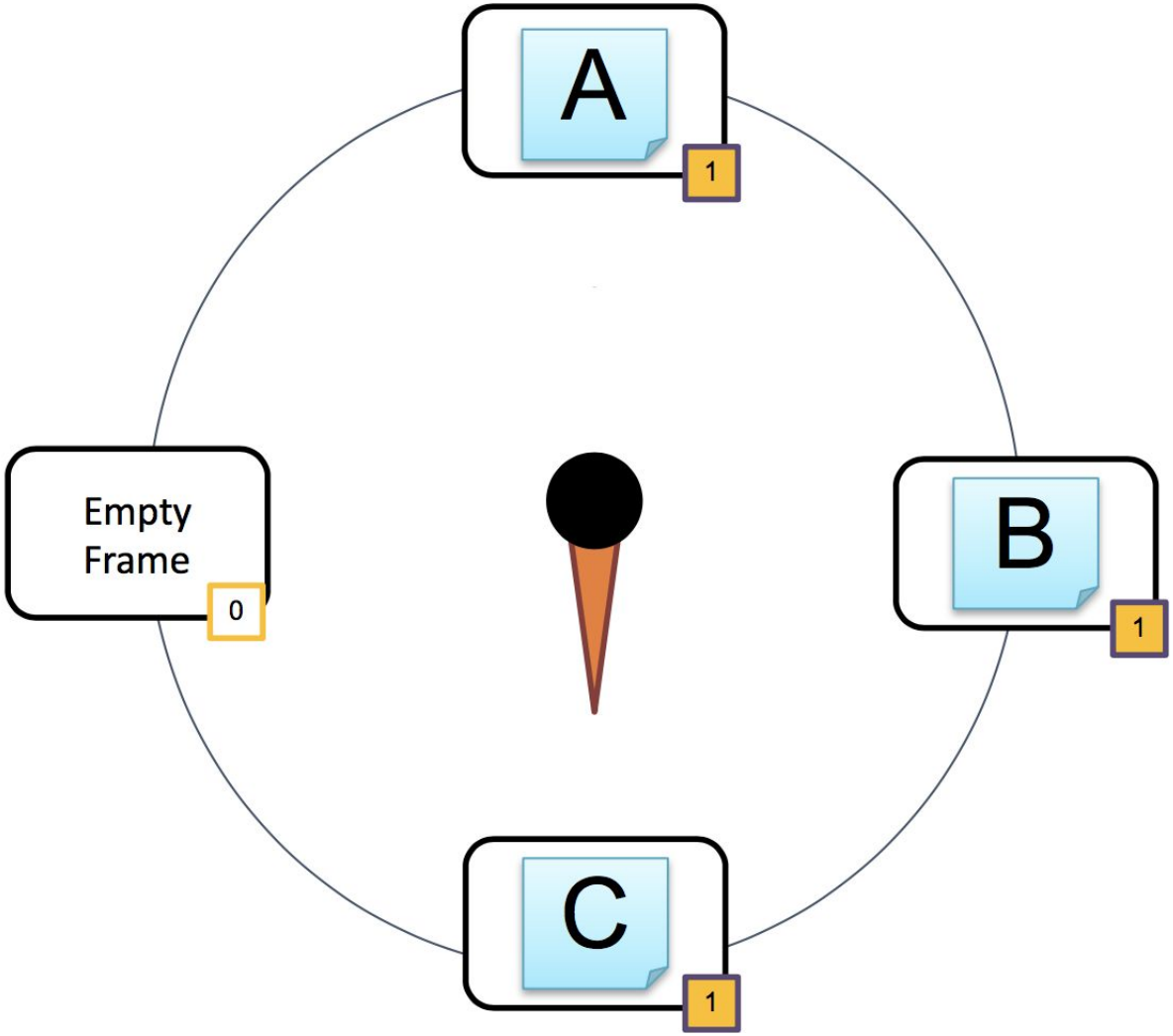
Buffer Management

Buffer Manager

- Sits between RAM (memory) and disk
- Controls when pages are read from and written to disk
- Page replacement policy helps decide which pages to evict
  - LRU (Least Recently Used): pin frames when in use, track when pages are last unpinned, kick out least recently pinned frame; takes advantage of temporal locality, but heap structure is expensive
  - MRU (Most Recently Used): similar to LRU, except kick out most recently pinned frame instead. Avoids sequential flooding (repeated scans causing needed pages to be kicked out)
- For problems, we usually assume pages are immediately unpinned after being pinned

Clock Policy

- More efficient approximation version of LRU (less metadata to store, still doesn't deal well with sequential flooding)
- Adds a clock hand that points to a frame, and a reference bit that is set to 1 when the page is used
- When inserting a page, look at the frame the clock hand is pointing at. If empty, or the reference bit is 0, evict page if necessary, then read in page, pin, set reference bit to 1, and move clock hand to next page
- If not empty, set reference bit of current frame to 0 and move clock hand to next page, and repeat until page is read in


A
1
Empty
Frame
0
B
1
C
1

Relational Algebra

Operators

- Every operator takes in relation(s), and returns a relation
- Set semantics (no duplicates)
- Unary operators (one relation):
  - Selection ($\sigma_{condition}$): Selects a subset of tuples matching some predicate (horizontal), similar to SQL's WHERE
  - Projection ($\pi_{col\ 1, ..., col\ n}$): Retains only desired attributes (vertical), similar to SQL's SELECT
  - Renaming ($\varrho_{a \to b}$): Rename attributes, similar to SQL's AS
- Binary operators (two relations):
  - Union (R ∪ S): Returns a relation containing all the tuples from both input relations, similar to SQL's UNION
  - Set Difference (R — S): Returns a relation containing all the tuples from the left relation, except for the ones appearing in the right relation, similar to SQL's EXCEPT
  - Cross Product (R × S): Returns the cross product of two relations (one tuple for every possible pair of tuples), similar to SELECT ... FROM R, S in SQL (no predicates)

Operators (cont.)

- Compound operators (combinations of other operators)
  - Intersection (R ∩ S): returns a relation that has only tuples appearing in both tables, similar to SQL's INTERSECT, equivalent to R — (R — S)
  - Theta Join ($\bowtie_\theta$): Returns the inner join of two relations on the specified join condition θ, similar to SQL's INNER JOIN, equivalent to $\sigma_\theta(R \times S)$
  - Natural Join ($\bowtie$): Returns the natural join of two relations (oined together on *every* column with the same name), similar to SQL's NATURAL JOIN, equivalent to $\sigma_{R.col1=S.col1 \wedge \ldots \wedge R.colN=S.colN}(R \times S)$
- Group By / Aggregation ($\gamma_{col,\ cond}(S)$): returns S grouped by col, and filters groups that don't meet cond, similar to SQL's GROUP BY and HAVING

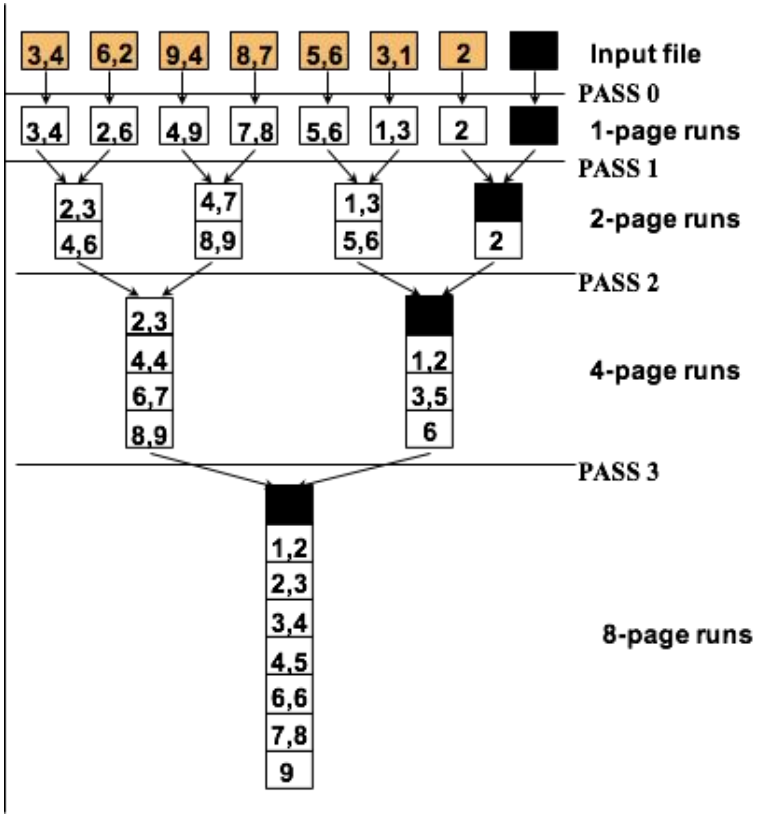
Misc. Notes

- Set semantics forces the condensing of duplicate tuples
  - Ex. projection if you remove differentiating columns, union (UNION vs UNION ALL), etc.
- Renaming useful for differentiating attributes before joins
- Union, set difference, intersection only defined when both input relations have the same attributes
- Cross product defined even when both relations have different attributes
- Can do nested queries using parentheses
  - $\pi_{R.a}(R) - \pi_{R.a}(R \bowtie_{R.a = S.b} S)$ like
    SELECT R.a FROM R WHERE R.a NOT IN (SELECT S.b FROM S)
  - $\pi_{R.a}(R) \cap \pi_{S.b}(dogs)$, and $\pi_{R.a}(cats \bowtie_{R.a = S.b} dogs)$, like
    SELECT R.a FROM R WHERE R.a IN (SELECT S.b FROM S)

Sorting/Hashing

Sorting

- We first sort small amounts of data into **runs** of sorted tuples
- Given runs of sorted tuples, we can **merge** them into 1 larger run of sorted tuples
  - Same as in-memory mergesort
  - Stream in the two runs and stream out the new run



General External Merge Sort

- How many passes do we need?
  - We sort B pages at once, so we have $\lceil N/B \rceil$ runs after Pass 0
  - We merge B-1 pages at once, so we have to do $\lceil \log_{B-1}(\#\ \text{runs}\ ) \rceil$ merge passes
  - So we have $1 + \lceil \log_{B-1}(\lceil N/B \rceil) \rceil$ passes over the data

Sorting

B buffer pgs -> merge
B-1 runs at a time

Initial
number of
sorted runs

$$2N * (1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil) \text{ I/Os.}$$

Read and write each
page per pass

Number of passes:
First pass + each merging pass

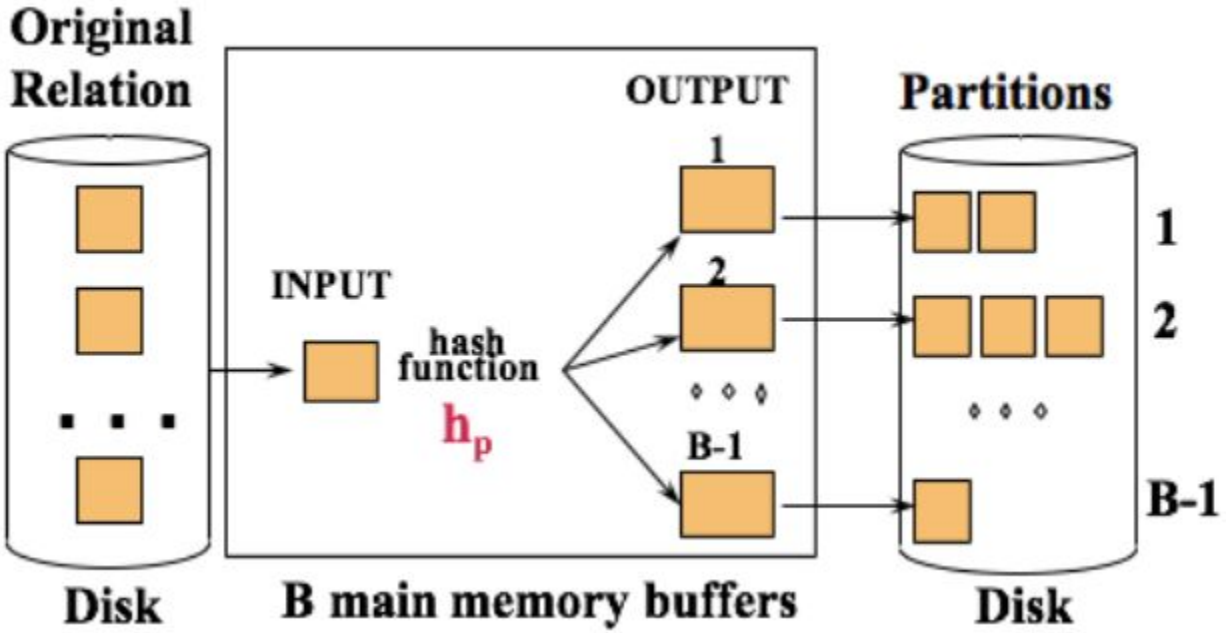
Sorting

- Two passes can sort B * (B-1) pages
- Three passes can sort B * $(B-1)^2$ pages
- P passes can sort B * $(B-1)^{P-1}$ pages

External Hashing

- We can’t build an in-memory hash table if there’s too much data!
- Start by splitting up data into smaller pieces!
  - Use a hash function h_p to partition the data
    - Stream partitions to disk
  - If we have B pages of buffer, we can split the data into B-1 partitions (1 buffer page reserved for streaming data in)

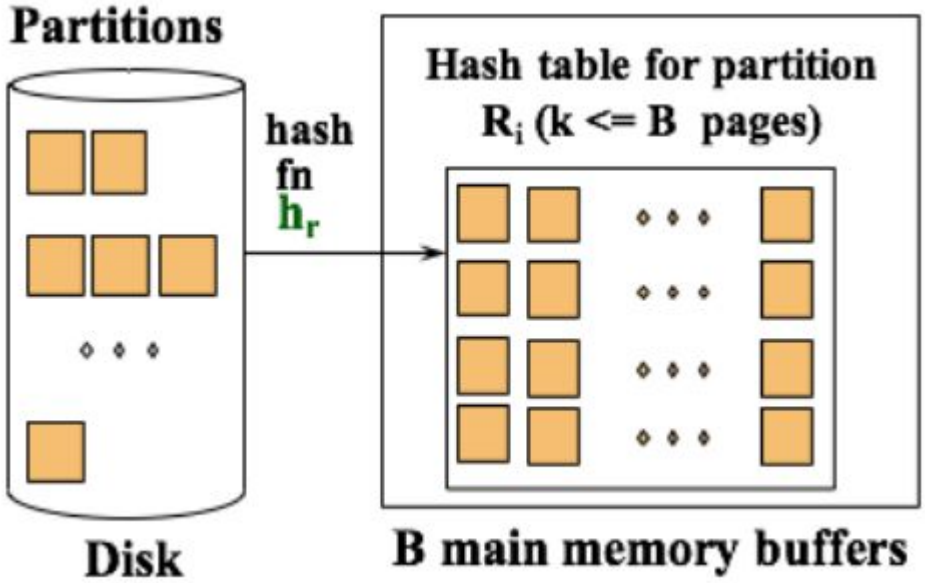
External Hashing



External Hashing

- If the partitions are small enough to fit in memory (at most B pages), we can load them in and make an in-memory hash table for each one, one at a time
  - Then we can apply duplicate removal, aggregation, etc. in memory
  - Every tuple in a partition has the same value when h_p is applied!
  - In-memory hash table must use a *different* hash function (we call it h_r) that is independent of h_p

External Hashing



Hashing

- **Important:** Don’t use the sorting formula for hashing
  - Each partition may not have the same number of pages

Hashing - Differs on Exams before Fall 2019

m - the total number of partitioning passes required

r_i - number of pages you need to read in for partitioning pass i

w_i - number of pages you need to write out of partitioning pass i

Factor of **2** is reading and writing each page

X - total number of pages we have after partitioning that we need to build our hash tables out of

$$\left(\sum_{i=1}^{m} r_i + w_i\right) + 2X$$

Refer to Note 7: Hashing

Worksheet

Solutions

Question 3

Question 3.1

Suppose the size of a page is 4KB, and the size of the memory buffer is 1 MB (1024 KB). We have a relation of size **800 KB**. How many page IOs are required to sort this relation?

Answer: 400.

800 KB with 4KB per page means 200 pages are needed to store the relation. Since this is small enough to completely fit into the buffer we only need to read it in, sort it (no I/Os required for sorting), then write the sorted pages back to disc.

200 (to read in) + 200 (to write out) = 400 I/Os

Question 3.2

Suppose the size of a page is 4KB, and the size of the memory buffer is 1 MB (1024 KB). We have a relation of size **5000 KB**. How many page IOs are required to sort this relation?

Answer: 5000.

5000 KB with 4KB per page means 1250 pages are needed to store the relation. We have 1024 / 4 = 256 pages in our buffer. That means the number of passes = $1 + \lceil \log_{255} \lceil 1250/256 \rceil \rceil = 2$

2 passes means two reads and two writes, which means there are 2 * 1250 + 2 * 1250 = 5000 I/Os.

Question 3.3

Suppose the size of a page is 4KB, and the size of the memory buffer is 1 MB (1024 KB). What is the size of the largest relation that would need two passes to sort?

Answer: 261,120 KB

Our buffer has room for 256 pages, so the maximum number of pages we can sort in two passes is **255 * 256 = 65,280.**

65,280 pages * 4 = **261,120 KB**

Question 3.4

Suppose the size of a page is 4KB, and the size of the memory buffer is 1 MB (1024 KB). What is the size of the largest relation we can possibly hash in two passes (i.e. with just one partitioning phase)?

Answer: 261,120 KB

Our buffer has room for 256 pages, so the maximum number of pages we can hash in two passes is **255 * 256 = 65,280.**

65,280 pages * 4 = **261,120 KB**

Question 3.5

Suppose we have a relation of size 3000 KB. We are executing a DISTINCT query on a column age, which has only two distinct values, evenly distributed. Would sorting or hashing be better here, and why?

Answer: Hashing, which allows us to remove duplicates early on and potentially improve performance (in this case, we might be able to finish in 1 pass, instead of 2 for sorting).

Question 3.6

Now suppose we were executing a GROUP BY on age instead. Would sorting or hashing be better here, and why?

Answer: Sorting because hashing won't work; each partition is larger than memory, so no amount of hash partitioning will suffice.

Question 4

Question 4.1

Assume our buffer pool has 8 frames. How many passes will it take to sort a 500 page file?

Answer: 4 passes.

Number of Passes = $1 + \lceil \log_{7} \lceil 500/8 \rceil \rceil = 4$

Question 4.2

Given the number of passes you calculated in 4.1, how many I/Os are necessary to externally sort the file?

Answer: 4000 I/Os.

2 * Number of Pages * Passes = 2 * 500 * 4 = 4000 I/Os.

Question 4.3

What is the minimum number of additional frames needed to reduce the number of passes found in 4.1 by 1?

Answer: 1 additional frame.

Given that we had 4 passes in 2.1, we need to calculate how many pages it will take to sort the relation in 3 passes.

$B(B-1)^2 >= 500$

B = 9 frames. 9 - 8 = 1 additional frame. I/Os.

Question 4.4

What is the minimum number of additional frames needed to sort the file in one pass?

Answer: 492 additional pages.

We need to fit the entire table into the buffer if we are to sort the file in one pass. Therefore we need 500 total pages, and 500 - 8 = 492 pages.

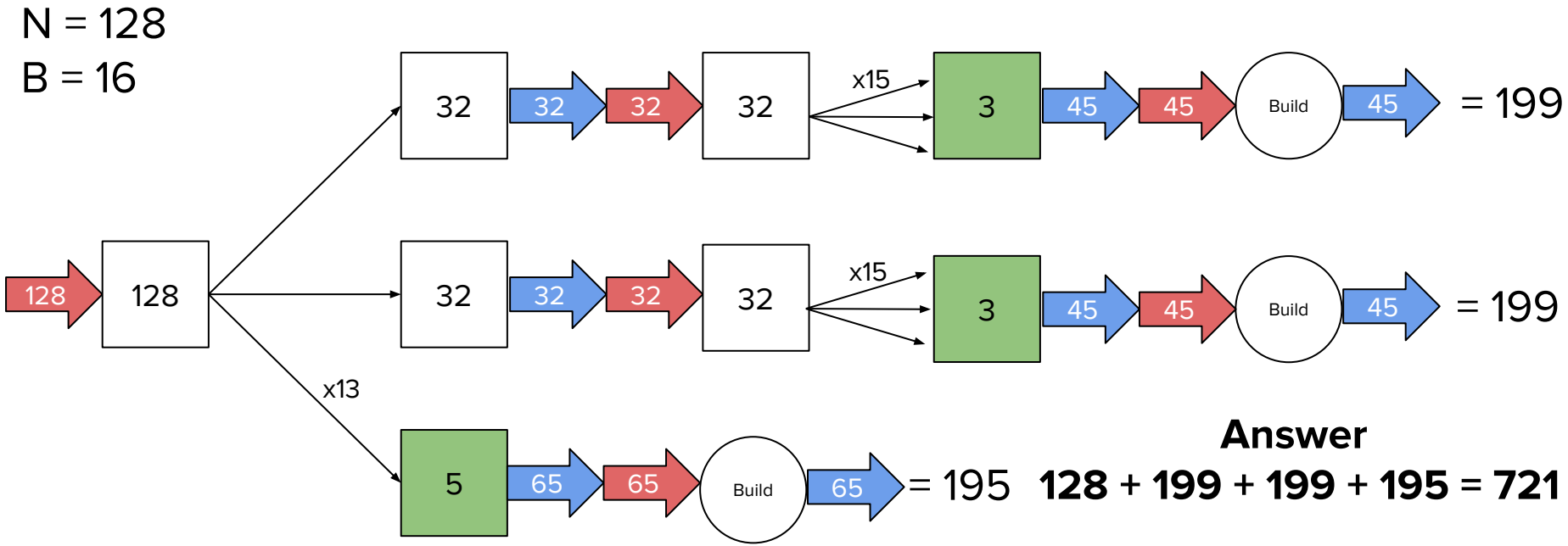
Question 5

Question 5.1

Suppose the size of each page is 4KB, and the size of our memory buffer is 64KB. What would be the I/O cost of hashing a file of 128 pages, assuming that the first hash function creates 2 partitions of 32 pages, and all other partitions are uniformly partitioned?

Answer: 721 pages.

Reads in red, writes in blue, green indicates can fit into buffers



Question 6

Question 6.1

Given an empty buffer pool with **4 pages** and the following access pattern:

A B C D E B A D C A E C

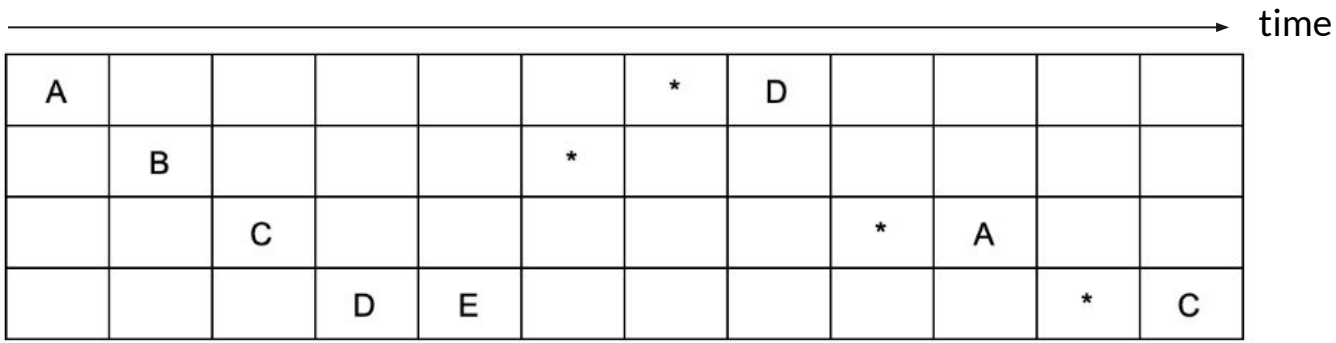
What is the hit rate for MRU?

Question 6.1

Given an empty buffer pool with **4 pages** and the following access pattern:

A B C D E B A D C A E C

What is the hit rate for MRU? Evict the Most Recently Used item → **4/12 hit rate**



Question 6.2

Given an empty buffer pool with **4 pages** and the following access pattern:

A B C D E B A D C A E C

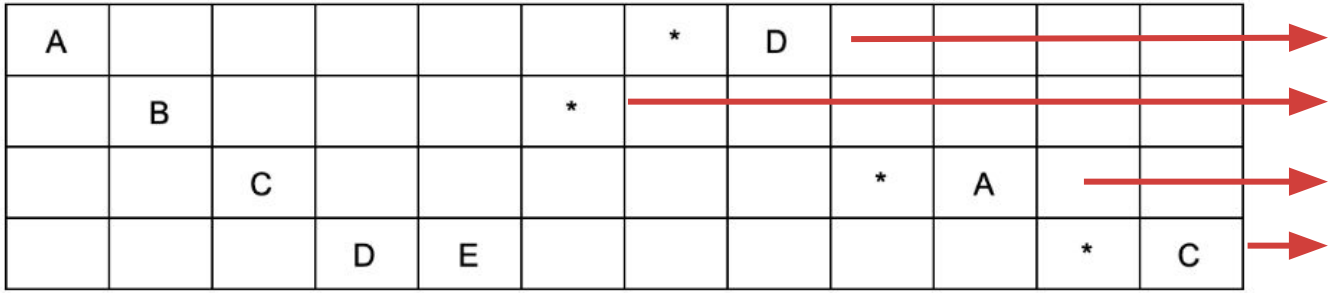
(Under MRU) in order of when they were *first* placed into the buffer pool, which pages remain in the buffer pool after this sequence of accesses?

Question 6.2

Given an empty buffer pool with **4 pages** and the following access pattern:

A B C D E B A D C A E C

(Under MRU) in order of when they were *first* placed into the buffer pool, which pages remain in the buffer pool after this sequence of accesses?



Answer:
B D A C

Question 6.3

Given an empty buffer pool with **4 pages** and the following access pattern:

A B C D E B A D C A E C

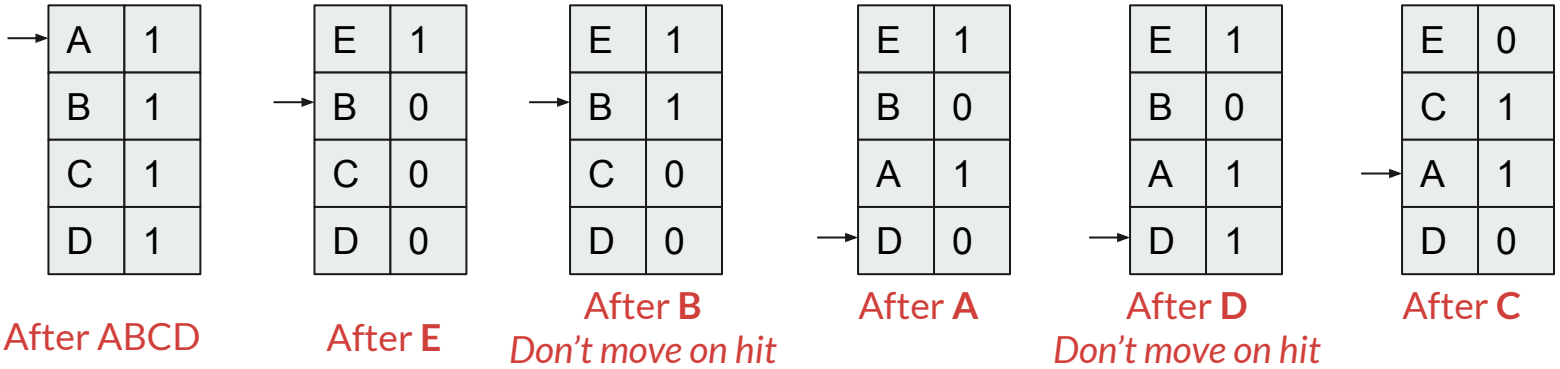
What is the hit rate for clock?

Question 6.3

Given an empty buffer pool with **4 pages** and the following access pattern:

A B C D E B A D C A E C

What is the hit rate for clock? **5/12**

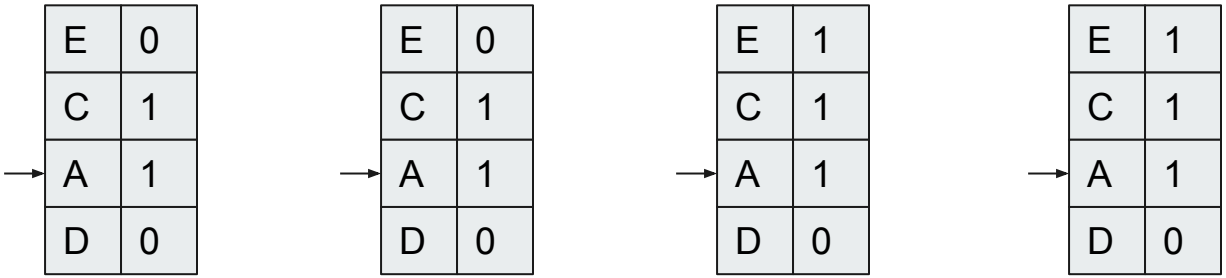


Question 6.3

Given an empty buffer pool with **4 pages** and the following access pattern:

A B C D E B A D C A E C

What is the hit rate for clock? **5/12**



After **C**
From previous slide

After **A**
Don't move on hit

After **E**
Don't move on hit

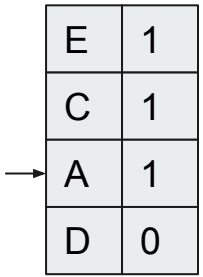
After **C**
Don't move on hit

Question 6.4

Given an empty buffer pool with **4 pages** and the following access pattern:

A B C D E B A D C A E C

Which pages are in the buffer pool after this sequence? **A C D E**



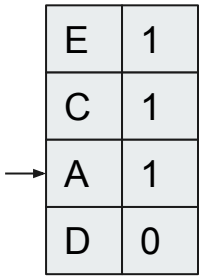
Final buffer pool

Question 6.5

Given an empty buffer pool with **4 pages** and the following access pattern:

A B C D E B A D C A E C

Which pages have their reference bit set? **A C E**



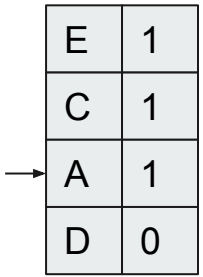
Final buffer pool

Question 6.6

Given an empty buffer pool with **4 pages** and the following access pattern:

A B C D E B A D C A E C

Which page is the arm of the clock pointing to? **A**



Final buffer pool

Question 7

Question 7.1

Consider two relations with the same schema: **R(A, B)** and **S(A, B)**

Which one of the following relational algebra expressions is not equivalent to the others?

(A) $\pi_{\mathrm{R.A}}((R \cup S) - S)$

(B) $\pi_{\mathrm{R.A}}(R) - \pi_{\mathrm{R.A}}(R \cap S)$

(C) $\pi_{\mathrm{R.A}}(R - S) \cap \pi_{\mathrm{R.A}}(R)$

(D) They are all equivalent.

Question 7.1

Consider two relations with the same schema: **R(A, B)** and **S(A, B)**

Which one of the following relational algebra expressions is not equivalent to the others?

(A) $\pi_{\mathrm{R.A}}((R \cup S) - S)$

(B) $\pi_{\mathrm{R.A}}(R) - \pi_{\mathrm{R.A}}(R \cap S)$

(C) $\pi_{\mathrm{R.A}}(R - S) \cap \pi_{\mathrm{R.A}}(R)$

(D) They are all equivalent.

Why?

A) Remove **S** tuples from the union of **R** and **S**

B)

C) Tuples in both **R-S** and **R**

Question 7.1

Consider two relations with the same schema: **R(A, B)** and **S(A, B)**

Which one of the following relational algebra expressions is not equivalent to the others?

(A) $\pi_{\mathrm{R.A}}((R \cup S) - S)$

(B) $\pi_{\mathrm{R.A}}(R) - \pi_{\mathrm{R.A}}(R \cap S)$

(C) $\pi_{\mathrm{R.A}}(R - S) \cap \pi_{\mathrm{R.A}}(R)$

(D) They are all equivalent.

Why?

A) Remove **S** tuples from the union of **R** and **S**

B) Remove **A** attrs common to **R** and **S** from **A** attrs in **R**

C) Tuples in both **R-S** and **R**

Issue if the same values for **A** in both relations:
R(A,B) = {(1, 2), (1, 3)} **S(A, B) = {(1, 3)}**
A: 1 tuple, B: 0 tuples, C: 1 tuple

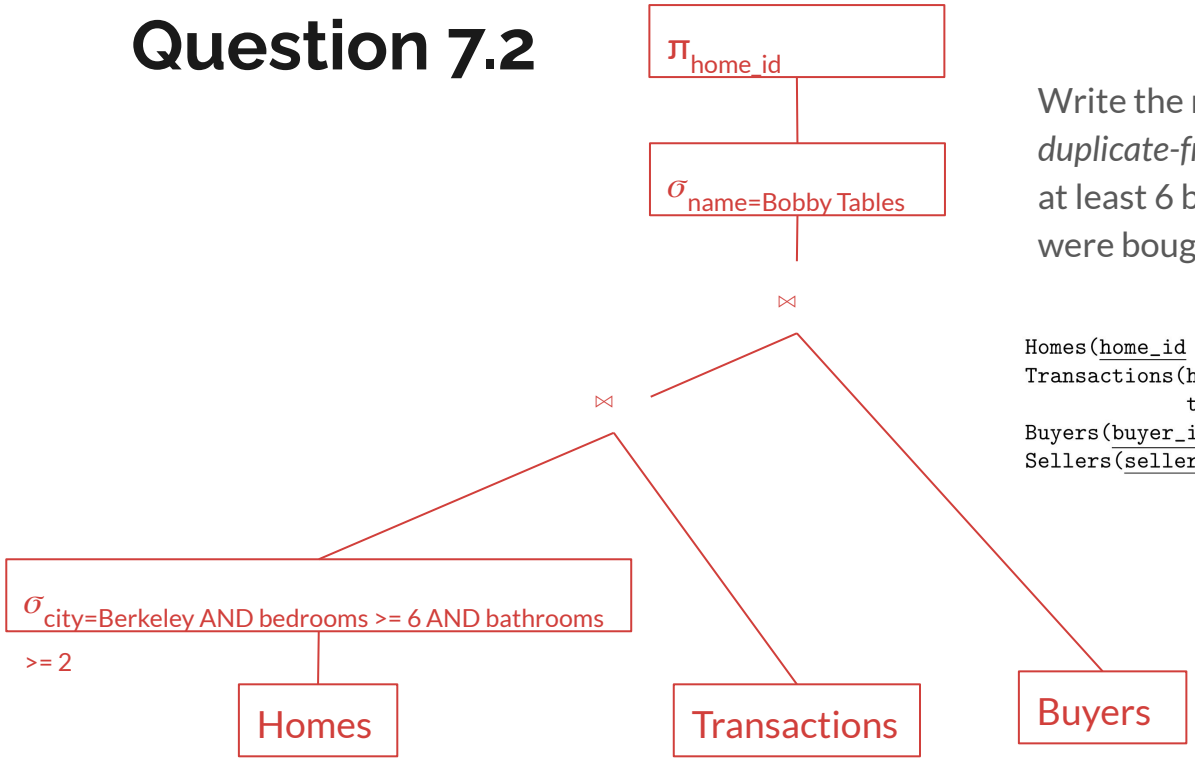
Question 7.2

The CS186 TAs have decided to go into real estate! All the information we need is described by the following schema (primary key of each table is underlined)

```
Homes(home_id int, city text, bedrooms int, bathrooms int, area int)
Transactions(home_id int, buyer_id int, seller_id int,
             transaction_date date, sale_price int)
Buyers(buyer_id int, name text)
Sellers(seller_id int, name text)
```

Write the relational algebra plan to find the *duplicate-free* set of IDs of all homes in Berkeley with at least 6 bedrooms and at least 2 bathrooms that were bought by Bobby Tables

Question 7.2



Write the relational algebra plan to find the *duplicate-free* set of IDs of all homes in Berkeley with at least 6 bedrooms and at least 2 bathrooms that were bought by Bobby Tables

```
Homes(home_id int, city text, bedrooms int, bathrooms int, area int)
Transactions(home_id int, buyer_id int, seller_id int,
             transaction_date date, sale_price int)
Buyers(buyer_id int, name text)
Sellers(seller_id int, name text)
```

Question 7.2

Write the relational algebra plan to find the *duplicate-free* set of IDs of all homes in Berkeley with at least 6 bedrooms and at least 2 bathrooms that were bought by Bobby Tables

```
Homes(home_id int, city text, bedrooms int, bathrooms int, area int)
Transactions(home_id int, buyer_id int, seller_id int,
             transaction_date date, sale_price int)
Buyers(buyer_id int, name text)
Sellers(seller_id int, name text)
```

$$\pi_{\text{home\_id}}(\sigma_{\text{name='Bobby Tables'}}(((\sigma_{\text{city='Berkeley' AND bedrooms}\geq 6\text{ AND bathrooms}\geq 2}\ \text{Homes}) \bowtie \text{Transactions}) \bowtie \text{Buyers}))$$

Fill out our feedback form!

https://tinyurl.com/CS186ExamPrep2FA20