CS150: Database & Datamining

Lecture 12: B+ Tree II & Relational Operators I

ShanghaiTech-SIST

Spring 2019

Acknowledgement: Slides are adopted from the Berkeley course CS186 by Joey Gonzalez and Joe Hellerstein, Stanford CS145 by Peter Bailis.

Announcements

- Mid-term 16th April
- Up to 8th week: Lecture 16

Today’s Lecture

1. B+ Trees - II

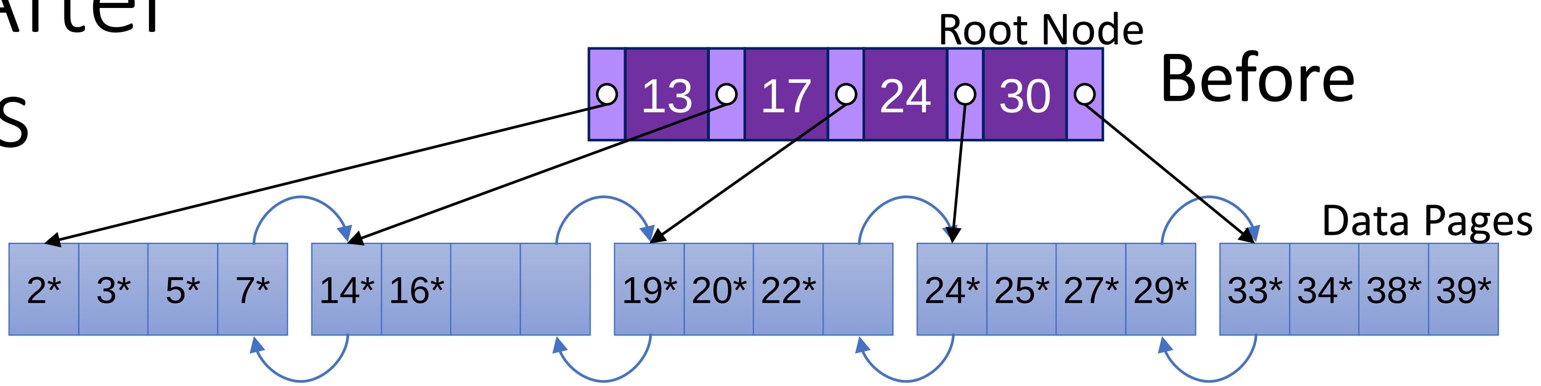
2. Relational Operators: Join algorithms

1. B+ Trees: Cost model

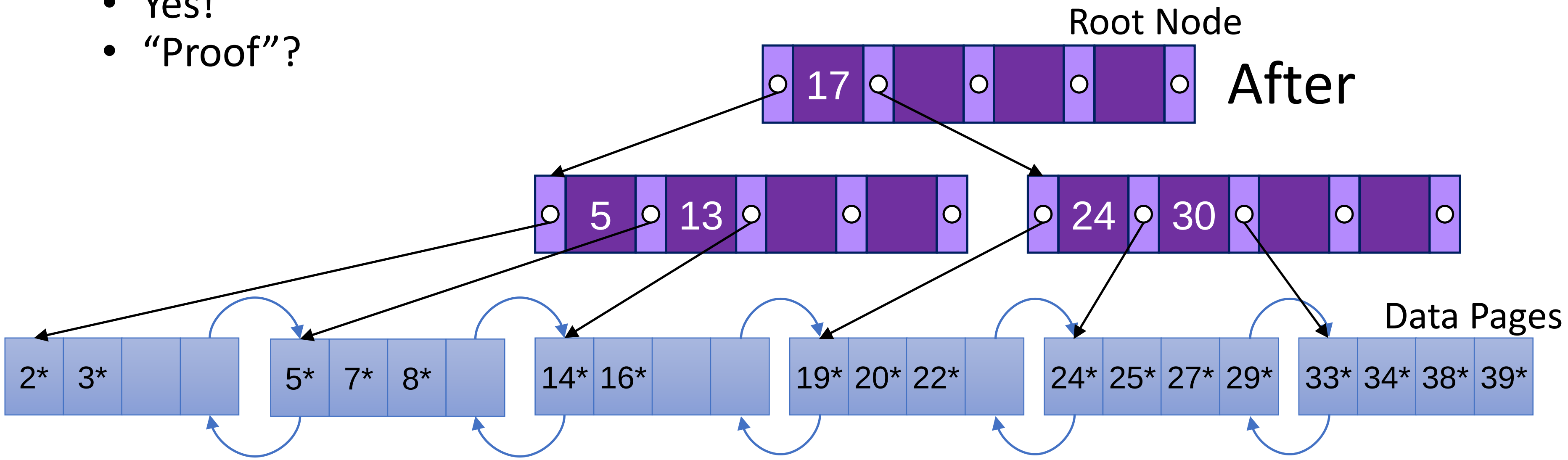
Inserting a Data Entry into a B+ Tree

- Find correct leaf *L.*
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Redistribute entries evenly, **copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - To split index node, redistribute entries evenly, but **push up** middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height.
  - Tree growth: gets *wider* or *one level taller at top.*

Before and After Observations



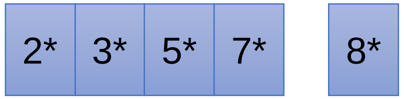
- Notice that the **root** was split to increase the height
  - *Grow from the root not the leaves*
  - ➔ All paths from root to leaves are equal lengths
- Does the occupancy invariant hold?
  - All nodes (except root) are at least half full
  - Yes!
  - “Proof”?



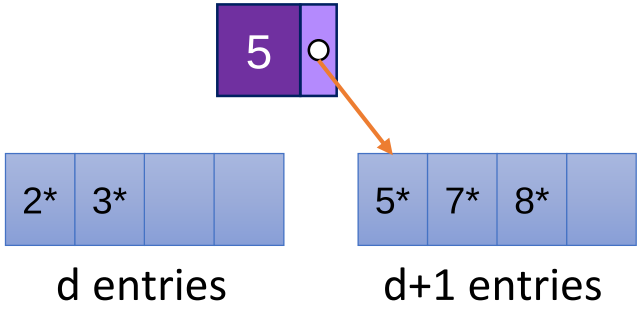
Splitting a Leaf

d = 2

- Start with full leaf (2d) entries:
  - Add a 2d +1 entry (8*)



- Split into leaves with (d, d+1) entries
  - **Copy** key up to parent



Why copy key and not push key up to parent?

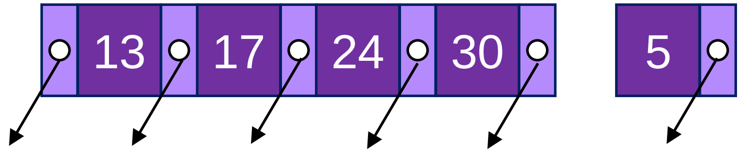
Key has value attached (5*)

Occupancy invariant holds after split.

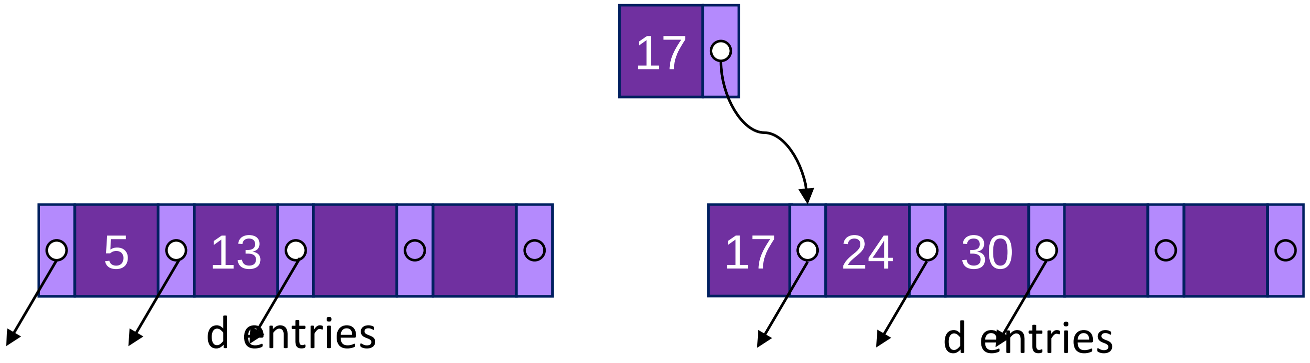
Splitting an interior node

d = 2

- Start with full interior node (2d) entries:
  - Add a 2d +1 entry



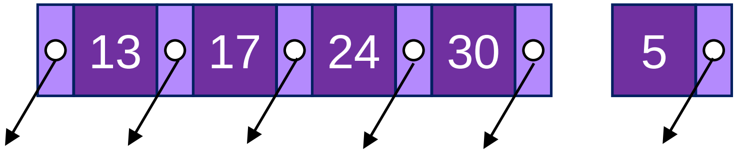
- Split into nodes with (d, d) entries
  - **Push** key up to parent



Splitting an interior node

d = 2

- Start with full interior node (2d) entries:
  - Add a 2d +1 entry

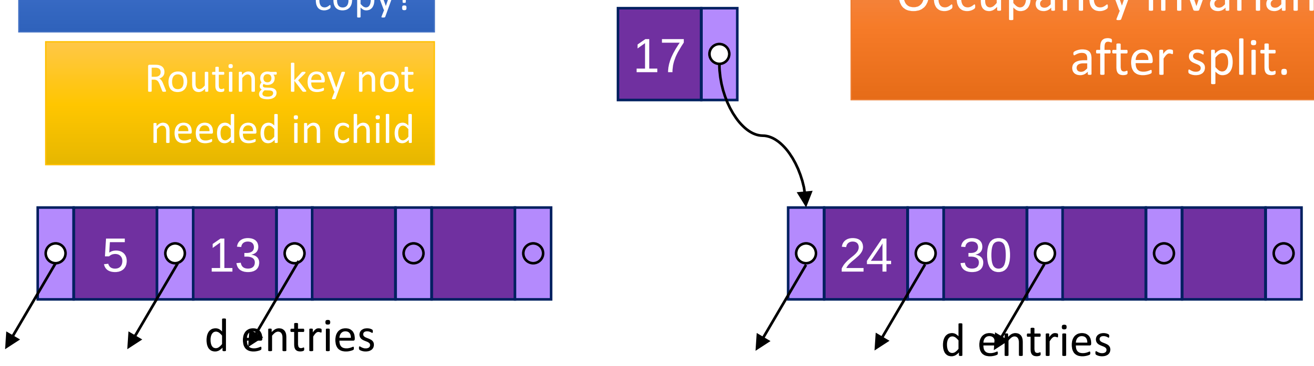


- Split into nodes with (d, d) entries
  - **Push** key up to parent

Why push and not copy?

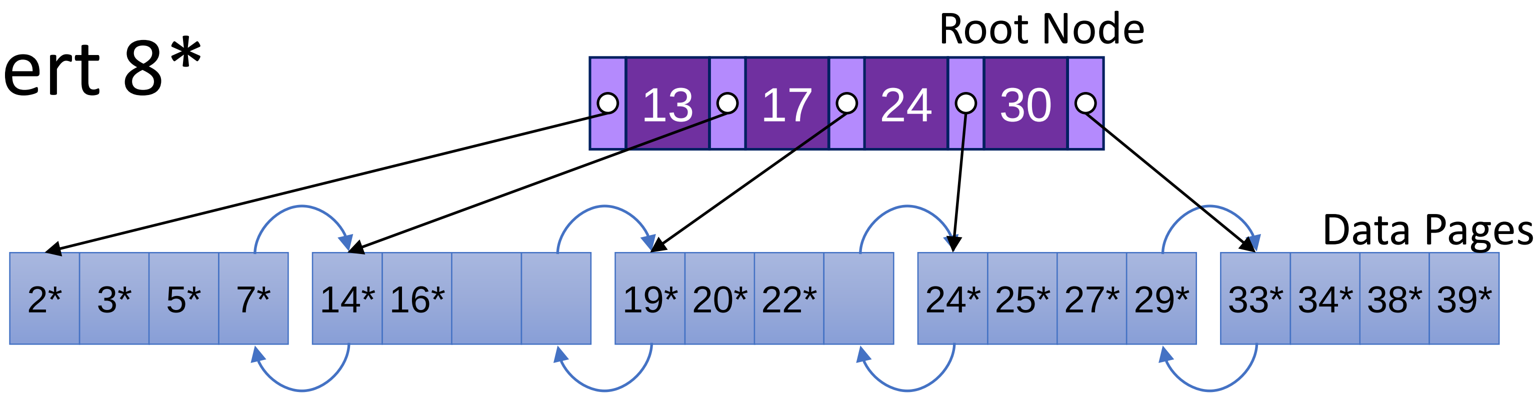
Routing key not needed in child

Occupancy invariant holds after split.

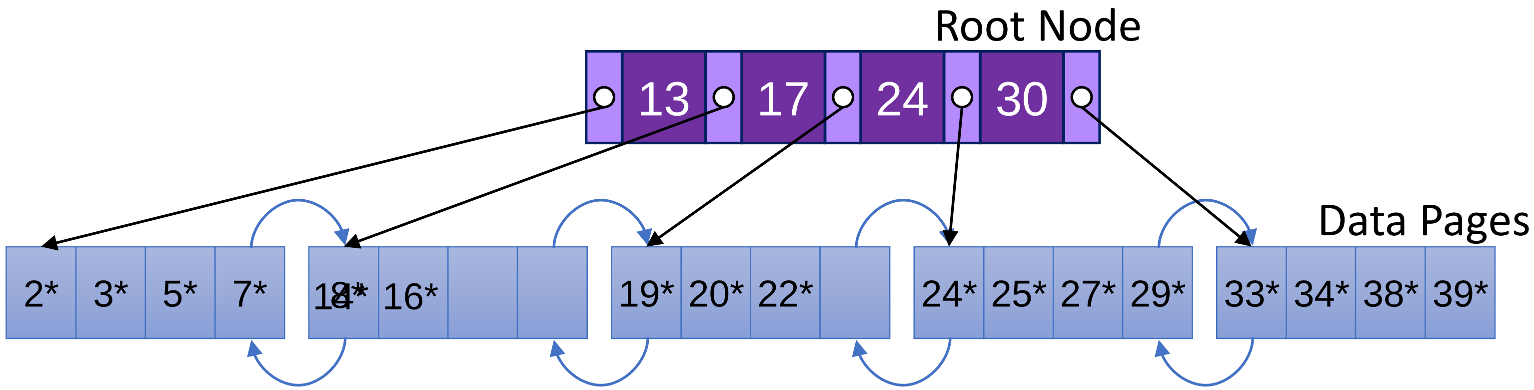


Did we have to split?

Insert 8*

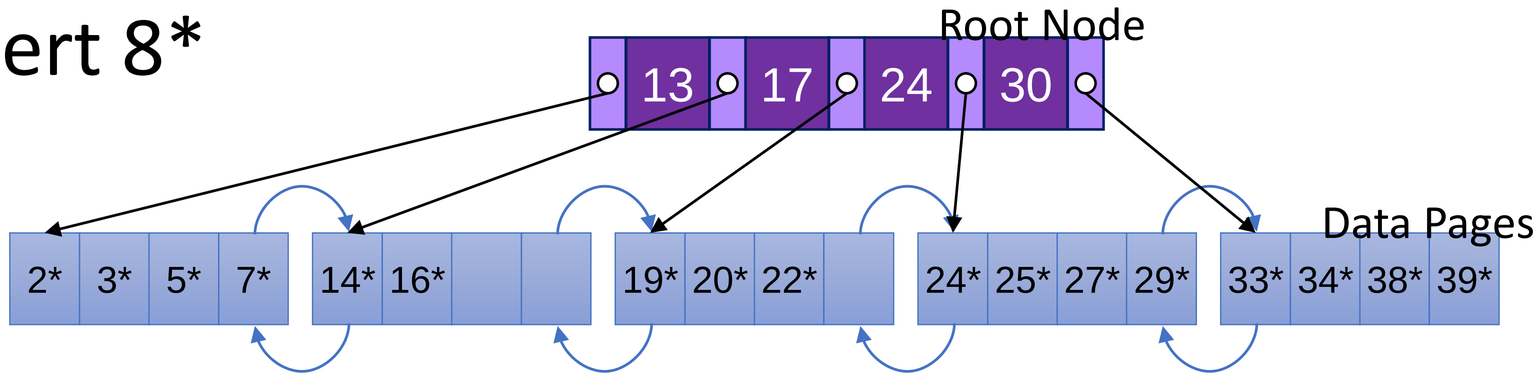


- What else could we do?
  - Redistribute keys?



Did we have to split?

Insert 8*

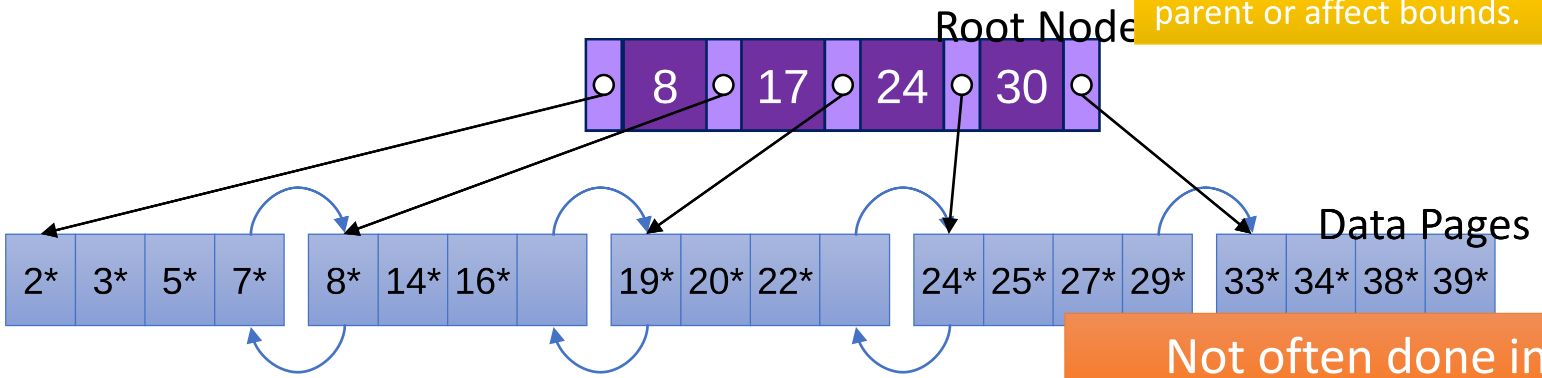


Do we need to recurse?

- What else could we do?
  - Redistribute keys?

No. Why?

Change does not split parent or affect bounds.



Not often done in practice

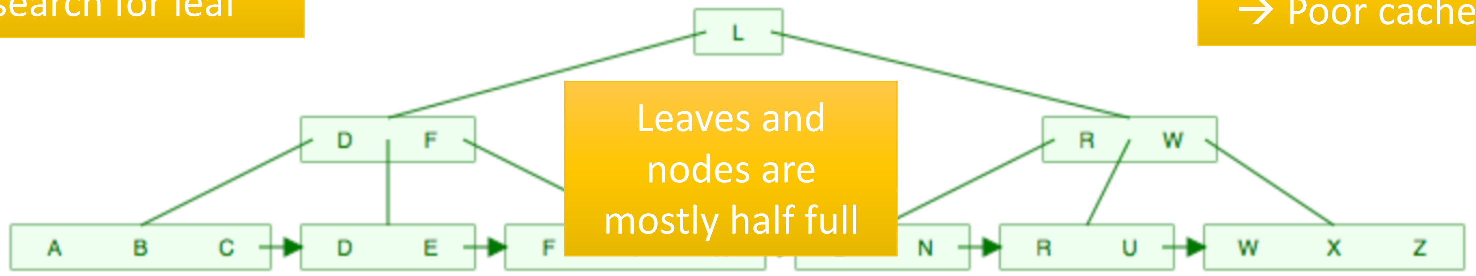
Bulk Loading of B+ Tree

Suppose we want to build an index on a large table

- Would it be efficient to just call insert repeatedly?
  - No … Why not?
  - Random Order: CLZARNDXEKFWIUB

Constantly need to search for leaf

Modifying random pages → Poor cache efficiency



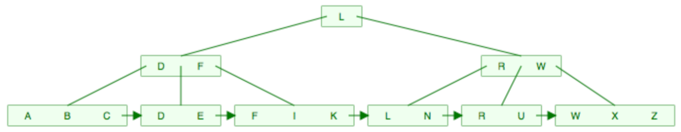
Leaves and nodes are mostly half full

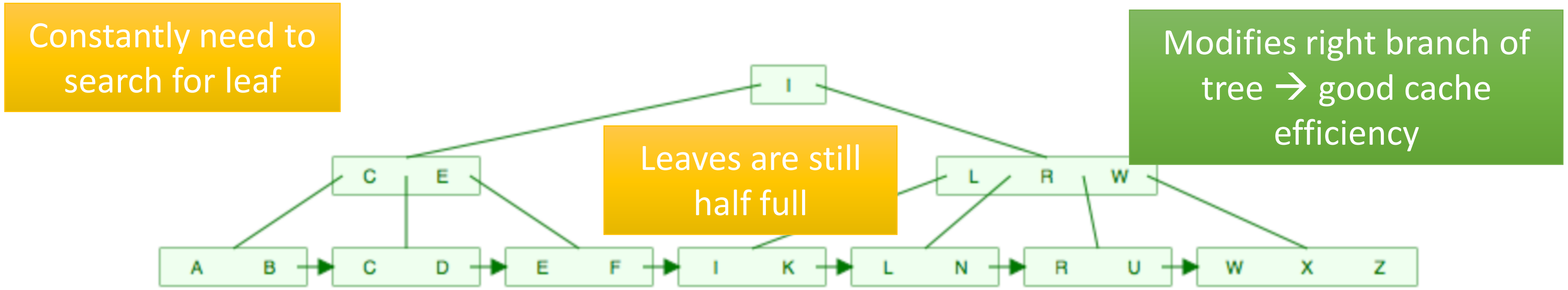
Work through this animation https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

Bulk Loading of B+ Tree

Suppose we want to build an index on a large table

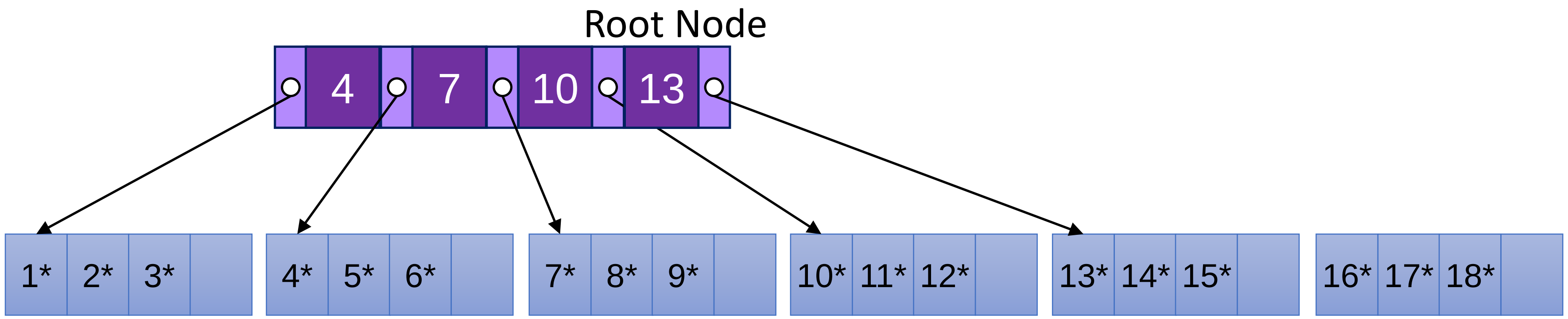
- Would it be efficient to just call insert repeatedly?
  - No … Why not?
  - **Sorterd** Order? ABCDEFIKLNRUWXZ





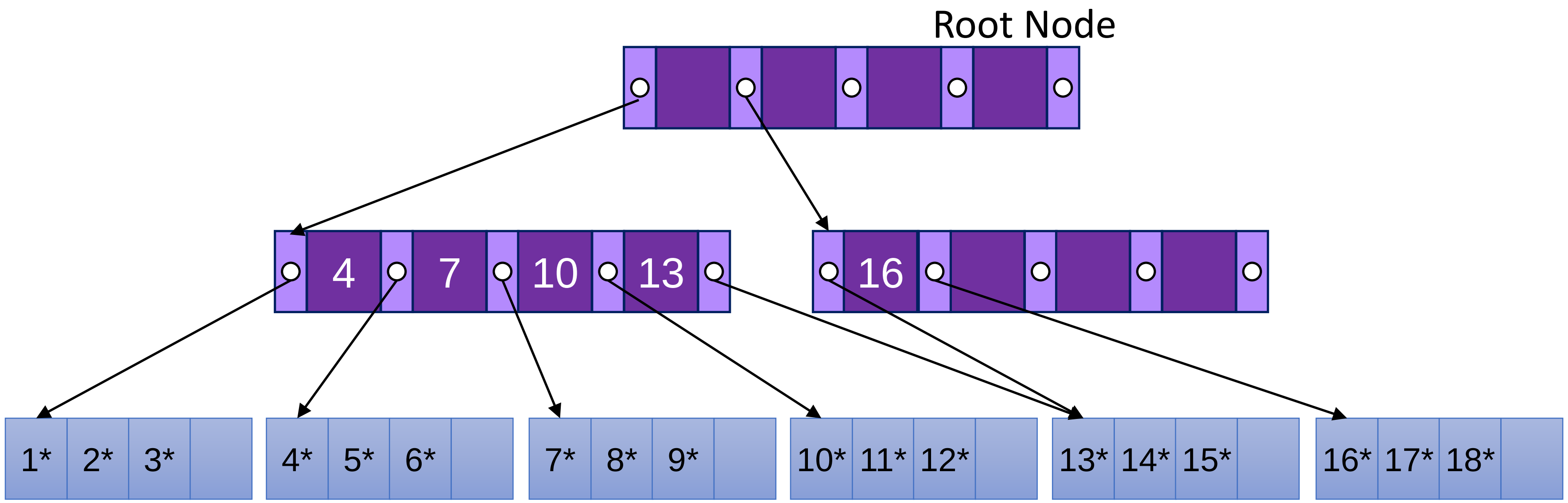
Smarter Bulk Loading a B+-Tree

- Sort the input records by key:
  - 1*, 2*, 3*, 4*, …
- Fill leaf pages to fill factor (e.g., ¾)
  - Updating parent until full



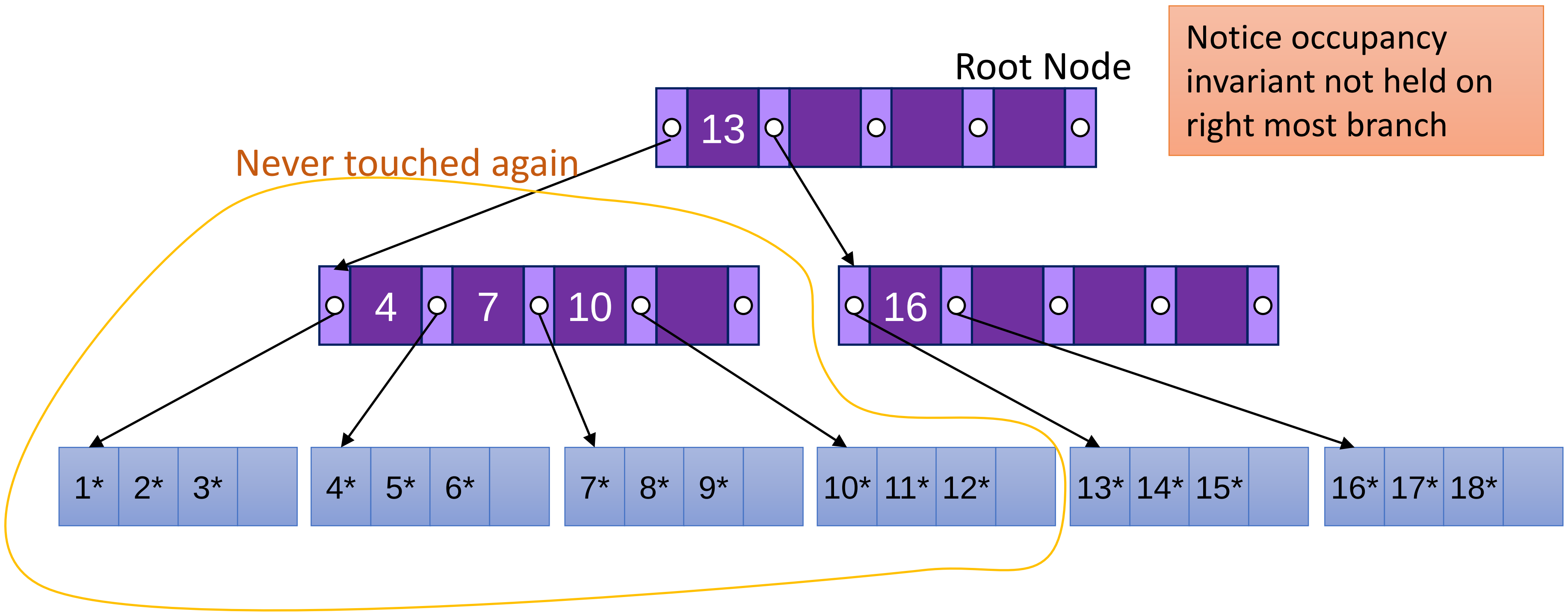
Smarter Bulk Loading a B+-Tree

- Sort the input records by key:
  - 1*, 2*, 3*, 4*, …
- Fill leaf pages to fill factor (e.g., ¾)
  - Updating parent until full
  - Split parent and copy to sibling to achieve fill factor



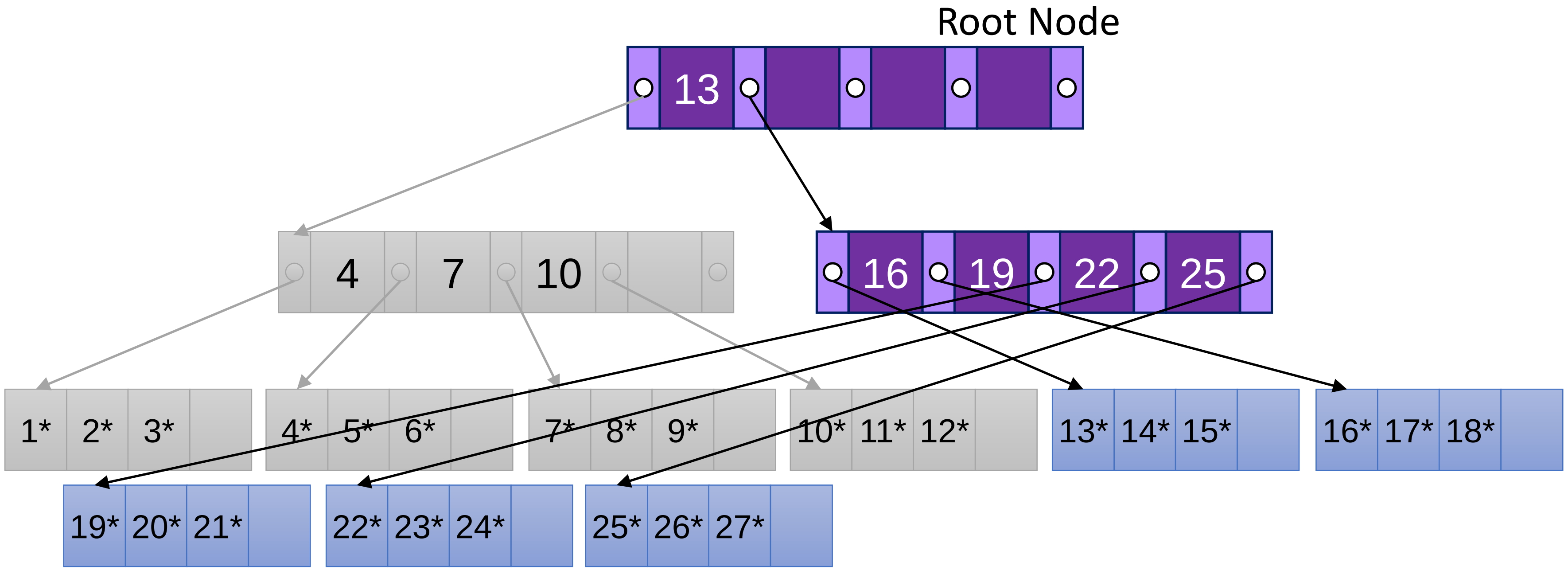
Smarter Bulk Loading a B+-Tree

- Sort the input records by key:
  - 1*, 2*, 3*, 4*, …
- Fill leaf pages to fill factor (e.g., ¾)
  - Updating parent until full
  - Split parent



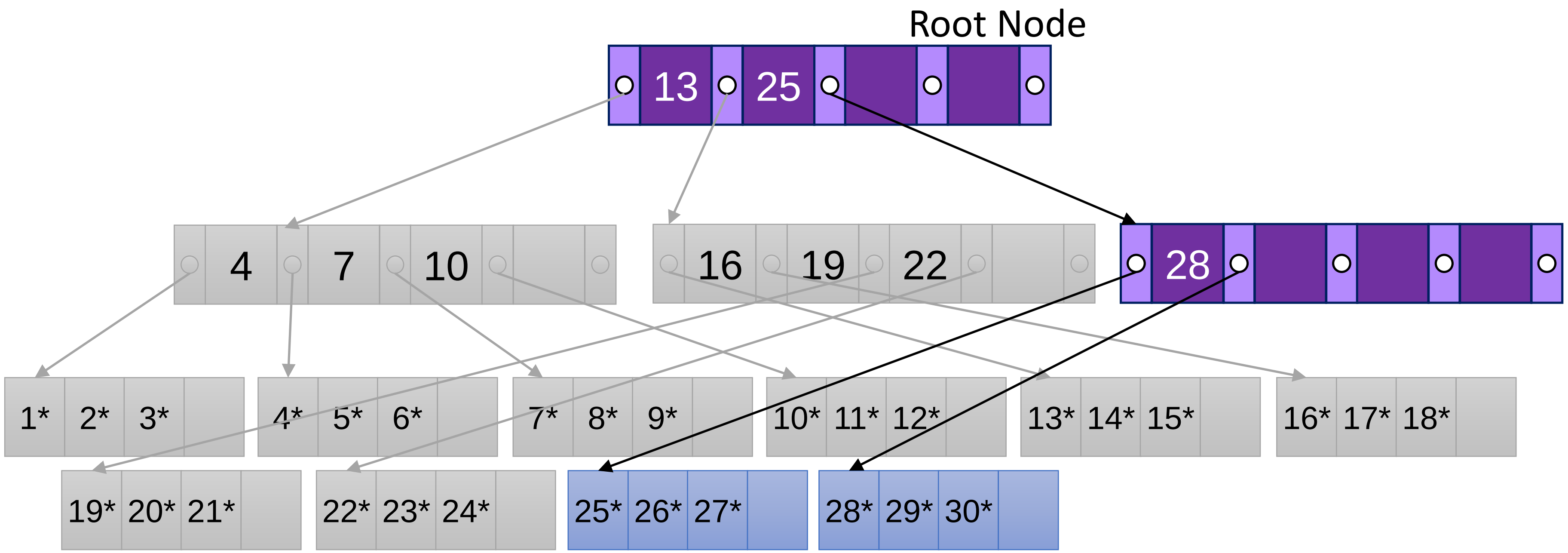
Smarter Bulk Loading a B+-Tree

- Sort the input records by key:
  - 1*, 2*, 3*, 4*, …
- Fill leaf pages to fill factor (e.g., ¾)
  - Updating parent until full
  - Split parent



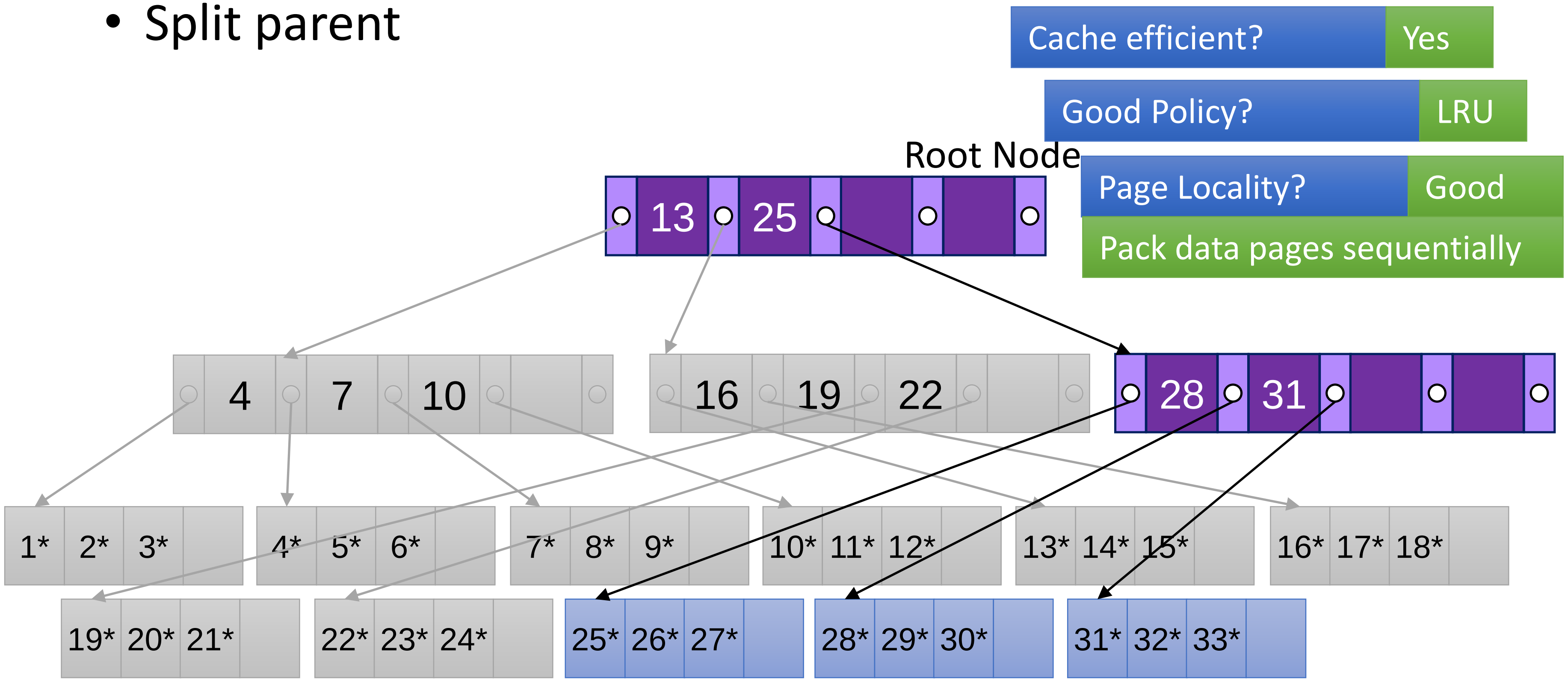
Smarter Bulk Loading a B+-Tree

- Sort the input records by key:
  - 1*, 2*, 3*, 4*, …
- Fill leaf pages to fill factor (e.g., ¾)
  - Updating parent until full
  - Split parent



Smarter Bulk Loading a B+-Tree

- Sort the input records by key:
  - 1*, 2*, 3*, 4*, …
- Fill leaf pages to fill factor (e.g., ¾)
  - Updating parent until full
  - Split parent



Summary of Bulk Loading

- Option 1: multiple inserts.
  - Slow.
  - Does not give sequential storage of leaves.

- Option 2: *Bulk Loading*
  - Fewer IOs during build. (Why?)
  - Leaves will be stored sequentially (and linked, of course).
  - Can control “fill factor” on pages.

Deleting a Data Entry from a B+ Tree

- Start at root, find leaf *L* where entry belongs.

- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has only **d-1** entries,
    - Try to re-distribute, borrowing from *sibling* *(adjacent node with same parent as L)*.
    - If re-distribution fails, *merge* *L* and sibling.

- If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.

- Merge could propagate to root, decreasing height.

You won't be tested on delete.

Cost of Operations

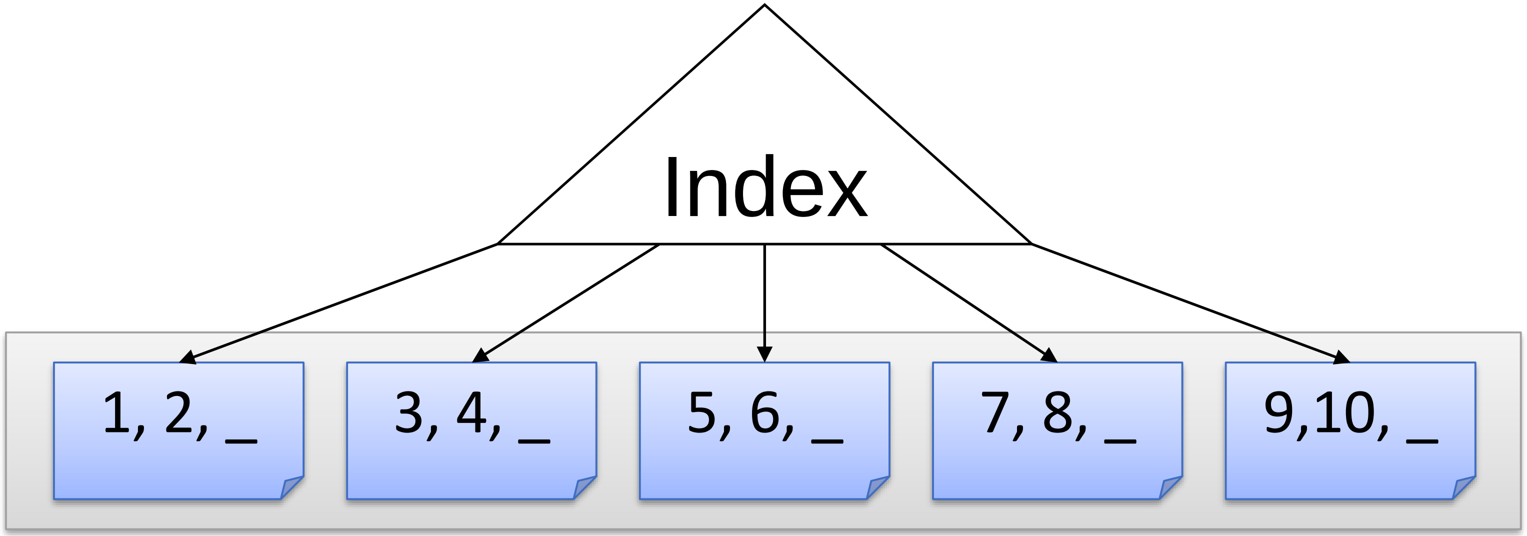
B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered Index
Scan all records	BD	BD	
Equality Search	0.5 BD	$(\log_2 B) * D$	
Range Search	BD	$[(\log_2 B) +$ #match pg$]*D$	
Insert	2D	$((\log_2 B)+B)D$	
Delete	0.5BD + D	$((\log_2 B)+B)D$	

Simple Clustered Index Analysis

Assumptions:

- Store data by reference (Alternative 2)
- Clustered tree index with 2/3 full heap file pages
  - Clustered → Heap file is initially sorted
  - **Fan-out** (F): relatively large. Why?
    - Page of <key, pointer> pairs ~ O(R)
  - Assume static index



Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered Index
Scan all records	BD	BD	?
Equality Search	0.5 BD	$(\log_2 B) * D$	
Range Search	BD	$[(\log_2 B) +$ #match pg$]*D$	
Insert	2D	$((\log_2 B)+B)D$	
Delete	0.5BD + D	$((\log_2 B)+B)D$	

Scan all the Records?

B: The number of data pages **(originally)**
R: Number of records per page **(originally)**
D: (Average) time to read or write disk page

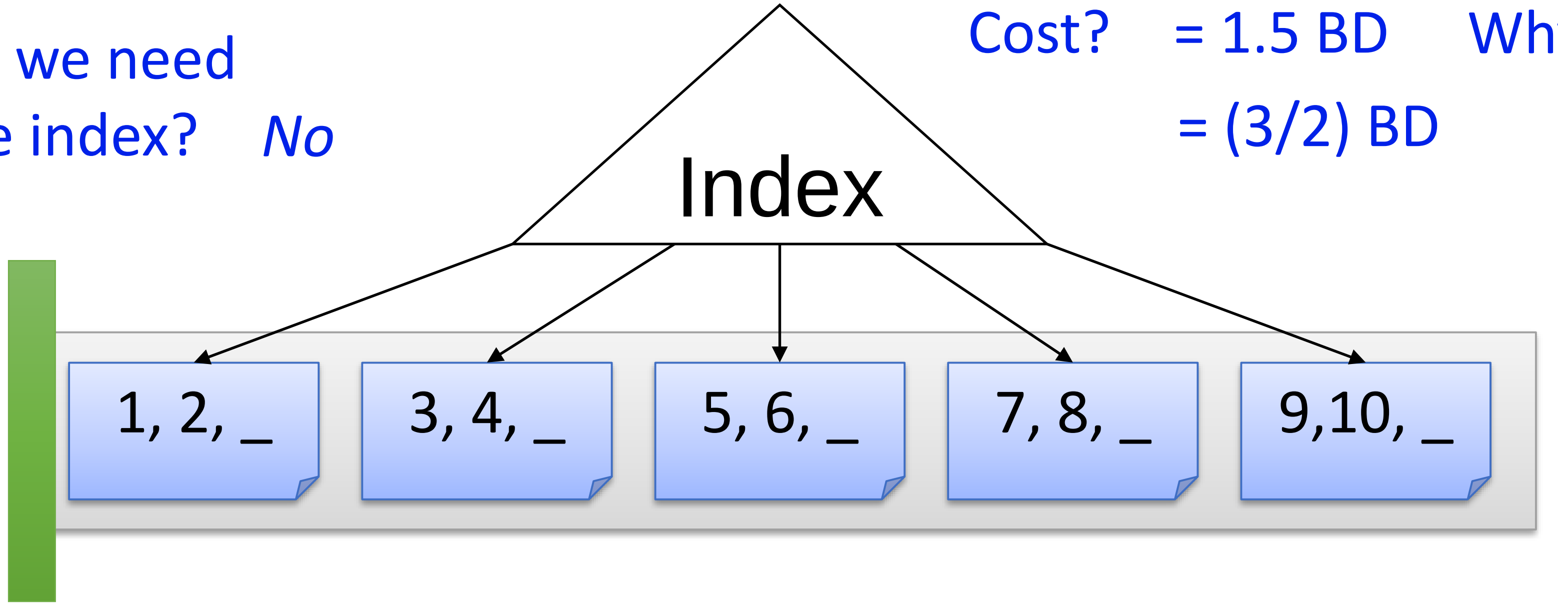
Assumptions:

- Store data by reference (Alternative 2)
- Clustered tree index with 2/3 full heap file pages
  - **Clustered** → Heap file is initially sorted
  - **Fan-out** (F): relatively large. ~ O(R)
  - Assume static index

Do we need the index? *No*

Cost? = 1.5 BD Why?

= (3/2) BD



Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered Index
Scan all records	BD	BD	1.5BD
Equality Search	0.5 BD	$(\log_2 B) * D$	
Range Search	BD	$[(\log_2 B) + \#\text{match pg}]*D$	
Insert	2D	$((\log_2 B)+B)D$	
Delete	0.5BD + D	$((\log_2 B)+B)D$	

Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered Index
Scan all records	BD	BD	1.5BD
Equality Search	0.5 BD	$(\log_2 B) * D$	?
Range Search	BD	$[(\log_2 B) +$ #match pg$]*D$	
Insert	2D	$((\log_2 B)+B)D$	
Delete	0.5BD + D	$((\log_2 B)+B)D$	

Find the record with key 3

Search the index:

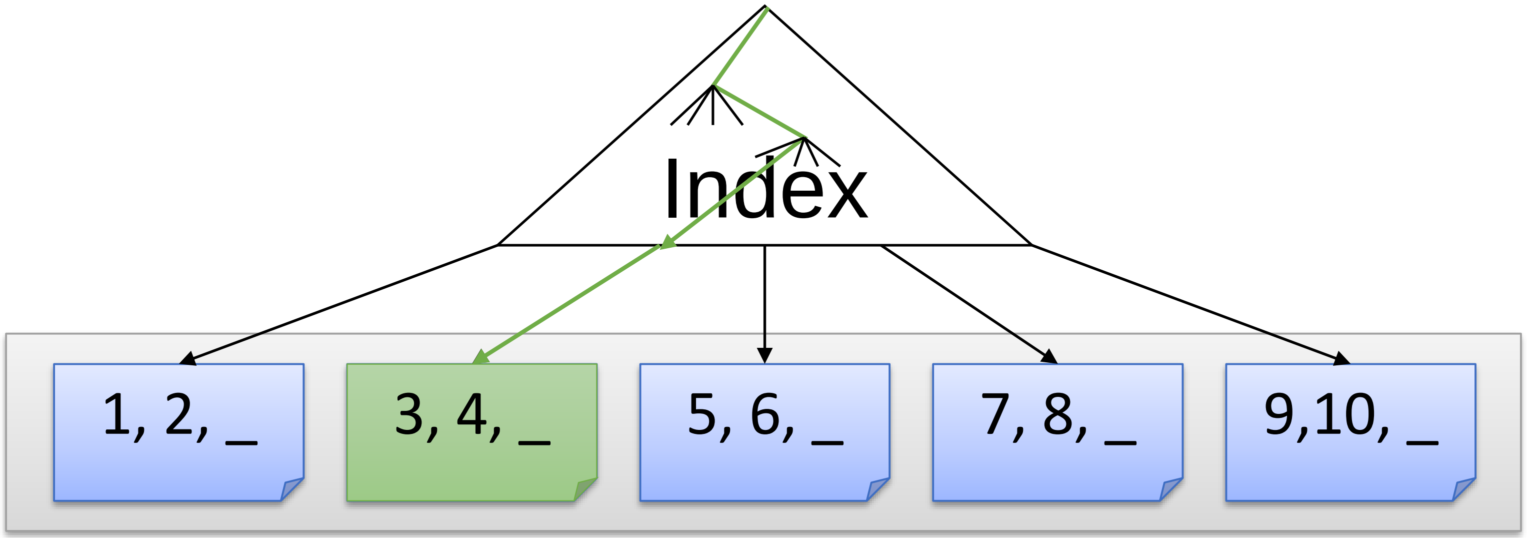
- Each page load narrows search by **factor of F**

Cost? $= \text{Log}_F(1.5\ B)\ D$

Lookup record in heap file by **record-id**

- Recall record-id = <page, slot #>

Cost? $=D$



Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered Index
Scan all records	BD	BD	(3/2) BD = 1.5BD
Equality Search	0.5 BD	$(\log_2 B) * D$	$(\log_F 1.5B+1) * D$
Range Search	BD	$[(\log_2 B) + \#\text{match pg}]*D$	
Insert	2D	$((\log_2 B)+B)D$	
Delete	0.5BD + D	$((\log_2 B)+B)D$	

Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered Index
Scan all records	BD	BD	(3/2) BD = 1.5BD
Equality Search	0.5 BD	$(\log_2 B) * D$	$(\log_F 1.5B+1) * D$
Range Search	BD	$[(\log_2 B) +$ #match pg$]*D$	?
Insert	2D	$((\log_2 B)+B)D$	
Delete	0.5BD + D	$((\log_2 B)+B)D$	

Find keys between 3 and 7

Search the index for 3: $= \mathrm{Log}_F(1.5\ B)\ D$

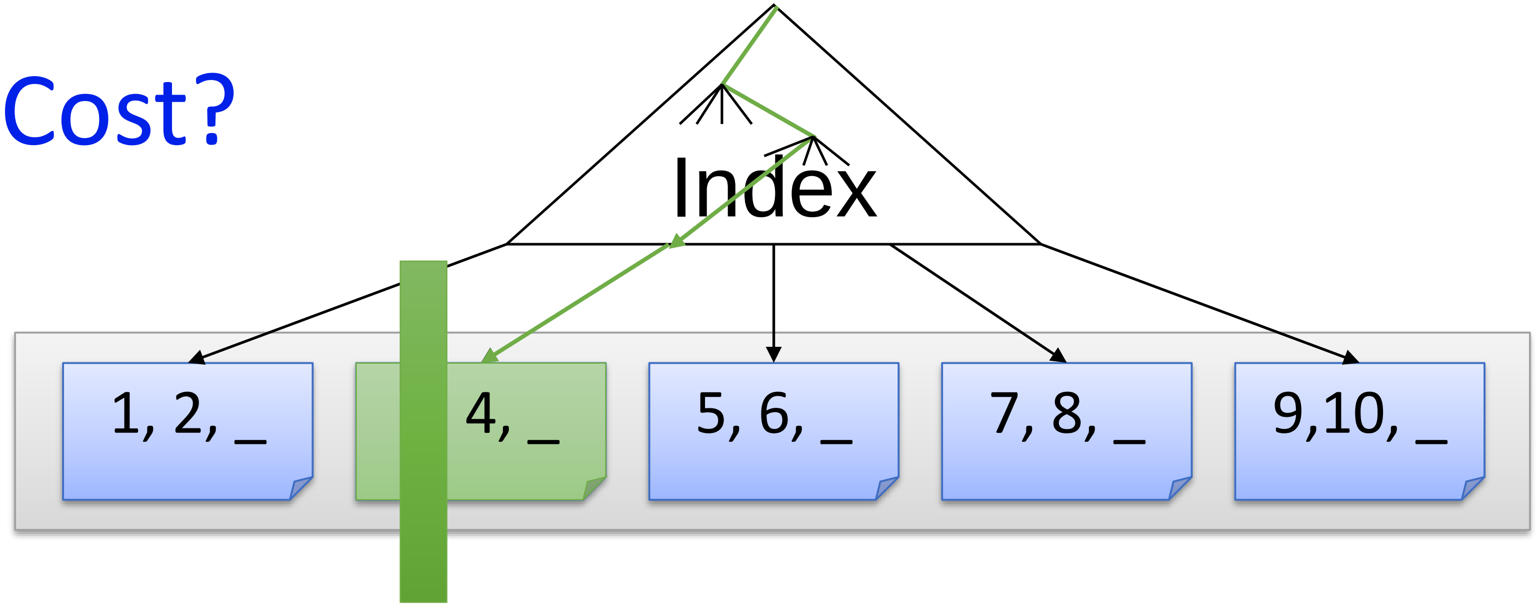
- Each page load narrows search by **factor of F**

Lookup record in heap file by **record-id** $= D$

- Recall record-id = <page, slot #>

Scan the leaves of index until the end of range (assume good clustering) $= (\#\text{match pg})\ D$

Cost?



Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered Index
Scan all records	BD	BD	(3/2) BD = 1.5BD
Equality Search	0.5 BD	$(\log_2 B) * D$	$(\log_F 1.5B+1) * D$
Range Search	BD	$[(\log_2 B) +$ #match pg$]*D$	$[(\log_F 1.5B) +$ #match pg$]*D$
Insert	2D	$((\log_2 B)+B)D$	
Delete	0.5BD + D	$((\log_2 B)+B)D$	

Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered Index
Scan all records	BD	BD	(3/2) BD = 1.5BD
Equality Search	0.5 BD	$(\log_2 B) * D$	$(\log_F 1.5B+1) * D$
Range Search	BD	$[(\log_2 B) +$ #match pg]*D	$[(\log_F 1.5B) +$ #match pg]*D
Insert	2D	$((\log_2 B)+B)D$	?
Delete	0.5BD + D	$((\log_2 B)+B)D$	

Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered Index
Scan all records	BD	BD	(3/2) BD = 1.5BD
Equality Search	0.5 BD	$(\log_2 B) * D$	$(\log_F 1.5B+1) * D$
Range Search	BD	$[(\log_2 B) +$ #match pg$]*D$	$[(\log_F 1.5B) +$ #match pg$]*D$
Insert	2D	$((\log_2 B)+B)D$	$((\log_F 1.5B)+2)*D$
Delete	0.5BD + D	$((\log_2 B)+B)D$	

Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered Index
Scan all records	BD	BD	(3/2) BD = 1.5BD
Equality Search	0.5 BD	$(\log_2 B) * D$	$(\log_F 1.5B+1) * D$
Range Search	BD	$[(\log_2 B) +$ #match pg]*D	$[(\log_F 1.5B) +$ #match pg]*D
Insert	2D	$((\log_2 B)+B)D$	$((\log_F 1.5B)+2)*D$
Delete	0.5BD + D	$((\log_2 B)+B)D$	

Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered Index
Scan all records	BD	BD	(3/2) BD = 1.5BD
Equality Search	0.5 BD	$(\log_2 B) * D$	$(\log_F 1.5B+1) * D$
Range Search	BD	$[(\log_2 B) +$ #match pg$]*D$	$[(\log_F 1.5B) +$ #match pg$]*D$
Insert	2D	$((\log_2 B)+B)D$	$((\log_F 1.5B)+2)*D$
Delete	0.5BD + D	$((\log_2 B)+B)D$	$((\log_F 1.5B)+2)*D$

Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	Sorted File	Clustered Index
Scan all records	BD	BD	(3/2) BD = 1.5BD
Equality Search	0.5 BD	$(\log_2 B) * D$	$(\log_F 1.5B+1) * D$
Range Search	BD	$[(\log_2 B) +$ #match pg]*D	$[(\log_F 1.5B) +$ #match pg]*D
Insert	2D	$((\log_2 B)+B)D$	$((\log_F 1.5B)+2)*D$
Delete	0.5BD + D	$((\log_2 B)+B)D$	$((\log_F 1.5B)+2)*D$

Summary

- We covered an algorithm + some optimizations for sorting larger-than-memory files efficiently
  - An ***IO aware*** algorithm!

- We create **indexes** over tables in order to support ***fast (exact and range) search*** and ***insertion*** over ***multiple search keys***

- **B+ Trees** are one index data structure which support very fast exact and range search & insertion via ***high fanout***
  - ***Clustered vs. unclustered*** makes a big difference for range queries too

2. Relational Operators

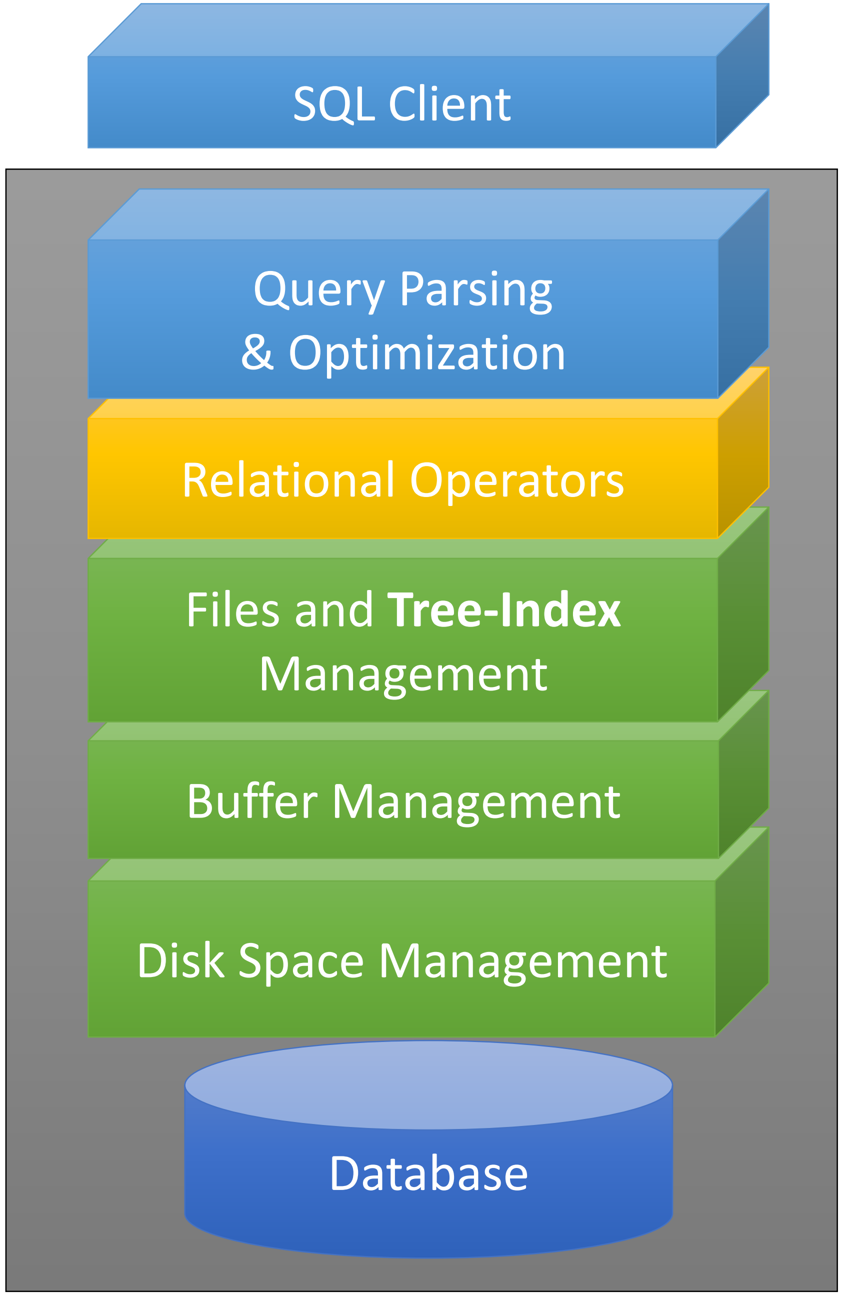
Architecture of a DBMS

Previously (2 Weeks)

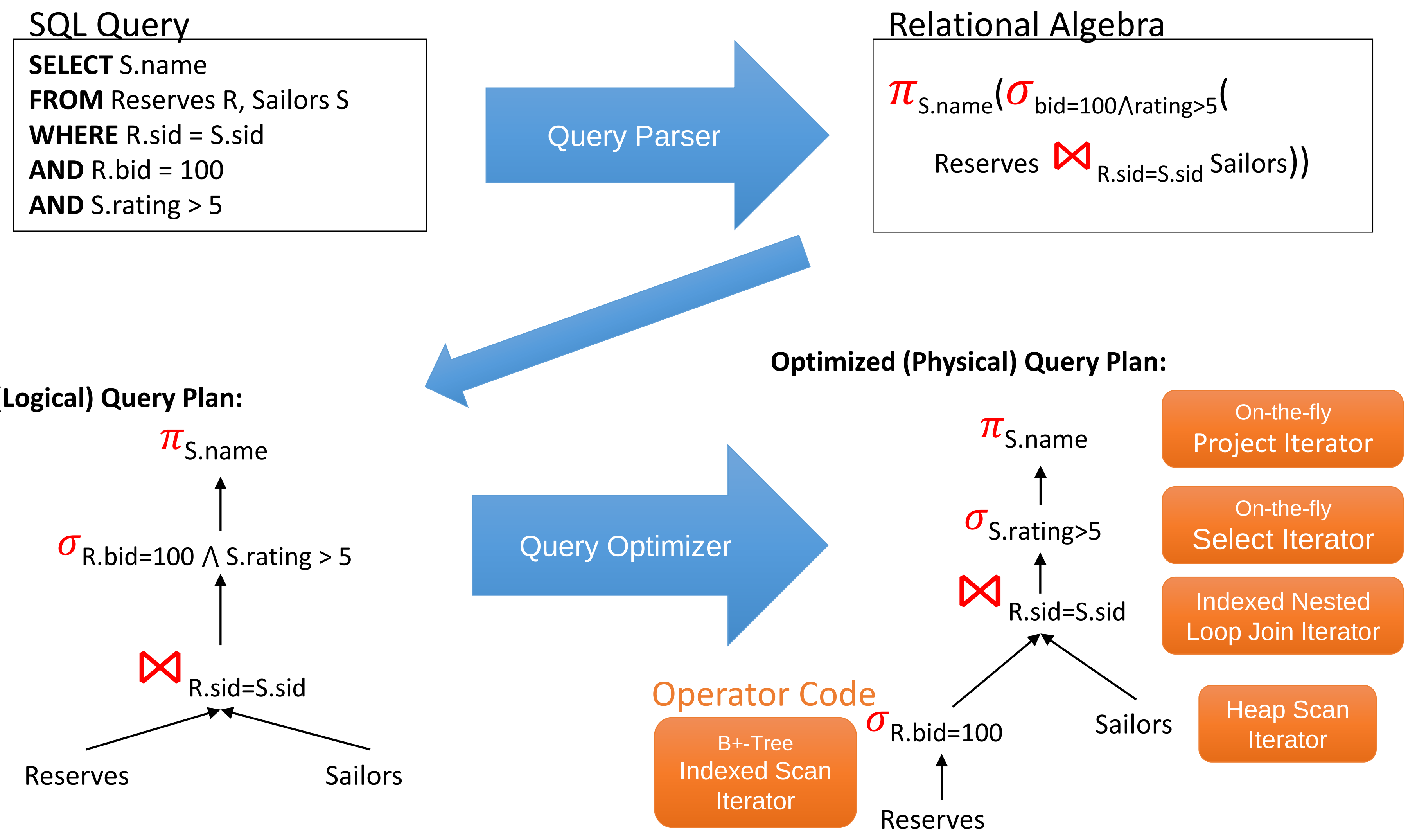
How do we **store** and **access** data?

Today

How de we **represent** and **execute** computation?



Big Picture Overview



Join Algorithms

A. Nested Loop Joins

B. Sort-Merge Join (next time)

C. Hash Joins (next time)

A. Nested Loop Joins

What you will learn about in this section

1. RECAP: Joins

2. Nested Loop Join (NLJ)

3. Block Nested Loop Join (BNLJ)

4. Index Nested Loop Join (INLJ)

RECAP: Joins

Joins: Example

$\mathbf{R} \bowtie \boldsymbol{S}$

```
SELECT R.A,B,C,D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples $\mathrm{r} \in R, s \in S$ *such that* $r.A = s.A$

R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

S

A	D
3	7
2	2
2	3

A	B	C	D
2	3	4	2

Joins: Example

$\mathbf{R} \bowtie \boldsymbol{S}$

```
SELECT R.A,B,C,D
FROM   R, S
WHERE  R.A = S.A
```

<u>Example:</u> Returns all pairs of tuples $\mathrm{r} \in R, s \in S$ *such that* $r.A = s.A$

R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

S

A	D
3	7
2	2
2	3

A	B	C	D
2	3	4	2
2	3	4	3

Joins: Example

$\mathbf{R} \bowtie \boldsymbol{S}$

```
SELECT R.A,B,C,D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples $\mathrm{r} \in R, s \in S$ *such that* $r.A = s.A$

R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

S

A	D
3	7
2	2
2	3

A	B	C	D
2	3	4	2
2	3	4	3
2	5	2	2

Joins: Example

$\mathbf{R} \bowtie \boldsymbol{S}$

```
SELECT R.A,B,C,D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples $\mathrm{r} \in R, s \in S$ *such that* $r.A = s.A$

R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

S

A	D
3	7
2	2
2	3

A	B	C	D
2	3	4	2
2	3	4	3
2	5	2	2
2	5	2	3

Joins: Example

$\mathbf{R} \bowtie \boldsymbol{S}$

```
SELECT R.A,B,C,D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples $\mathrm{r} \in R, s \in S$ *such that* $r.A = s.A$

R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

S

A	D
3	7
2	2
2	3

A	B	C	D
2	3	4	2
2	3	4	3
2	5	2	2
2	5	2	3
3	1	1	7

Semantically: A Subset of the Cross Product

$\mathbf{R} \bowtie \boldsymbol{S}$

```
SELECT R.A,B,C,D
FROM   R, S
WHERE  R.A = S.A
```

<u>Example:</u> Returns all pairs of tuples $\mathrm{r} \in R, s \in S$ *such that* $r.A = s.A$

R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

×

S

A	D
3	7
2	2
2	3

Cross Product …

Filter by conditions (r.A = s.A)

A	B	C	D
2	3	4	2
2	3	4	3
2	5	2	2
2	5	2	3
3	1	1	7

Can we actually implement a join in this way?

Notes

- We write $\mathbf{R} \bowtie \boldsymbol{S}$ to mean *join R and S by returning all tuple pairs where* ***all shared attributes*** *are equal*

- We write $\mathbf{R} \bowtie \boldsymbol{S}$ **on** ***A*** to mean *join R and S by returning all tuple pairs where* ***attribute(s) A*** *are equal*

- For simplicity, we'll consider joins on **two tables** and with **equality constraints** ("equijoins")

However joins *can* merge > 2 tables, and some algorithms do support non-equality constraints!

Nested Loop Joins

Notes

- We are again considering "IO aware" algorithms: ***care about disk IO***

- Given a relation R, let:
  - T(R) = # of tuples in R
  - P(R) = # of pages in R

Recall that we read / write entire pages with disk IO

- Note also that we omit ceilings in calculations… good exercise to put back in!

Nested Loop Join (NLJ)

```
Compute R ⋈ S on A:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

Nested Loop Join (NLJ)

Compute $\mathrm{R} \bowtie S \; on \; A$:

```
for r in R:
  for s in S:
    if r[A] == s[A]:
      yield (r,s)
```

Cost:

P(R)

1. **Loop over the tuples in R**

Note that our IO cost is based on the number of ***pages*** loaded, not the number of tuples!

Nested Loop Join (NLJ)

```
Compute R ⋈ S on A:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

Cost:

P(R) + T(R)*P(S)

1. Loop over the tuples in R
2. **For every tuple in R, loop over all the tuples in S**

Have to read *all of S* from disk for *every tuple in R!*

Nested Loop Join (NLJ)

```
Compute R ⋈ S on A:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

Cost:

P(R) + T(R)*P(S)

1. Loop over the tuples in R
2. For every tuple in R, loop over all the tuples in S
3. **Check against join conditions**

Note that NLJ can handle things other than equality constraints… just check in the *if* statement!

Nested Loop Join (NLJ)

```
Compute R ⋈ S on A:
  for r in R:
    for s in S:
      if r[A] == s[A]:
        yield (r,s)
```

What would ***OUT*** be if our join condition is trivial (*if TRUE*)?

OUT could be bigger than P(R)*P(S)… but usually not that bad

Cost:

P(R) + T(R)*P(S) + OUT

1. Loop over the tuples in R
2. For every tuple in R, loop over all the tuples in S
3. Check against join conditions
4. **Write out (to page, then when page full, to disk)**

Nested Loop Join (NLJ)

Compute $\mathrm{R} \bowtie S\ on\ A$:

```
for r in R:
  for s in S:
    if r[A] == s[A]:
      yield (r,s)
```

<u>Cost:</u>

P(R) + T(R)*P(S) + OUT

What if R (“outer”) and S (“inner”) switched?

P(***S***) + T(***S***)*P(***R***) + OUT

Outer vs. inner selection makes a huge difference- DBMS needs to know which relation is smaller!

IO-Aware Approach

Block Nested Loop Join (BNLJ)

Given ***B+1*** pages of memory

```
Compute R ⋈ S on A:
  for each B-1 pages pr of R:
    for page ps of S:
      for each tuple r in pr:
        for each tuple s in ps:
          if r[A] == s[A]:
            yield (r,s)
```

Cost:

$\mathrm{P}(R)$

1. **Load in B-1 pages of R at a time (leaving 1 page each free for S & output)**

Note: There could be some speedup here due to the fact that we're reading in multiple pages sequentially however we'll ignore this here!

Block Nested Loop Join (BNLJ)

Given ***B+1*** pages of memory

```
Compute R ⋈ S on A:
  for each B-1 pages pr of R:
    for page ps of S:
      for each tuple r in pr:
        for each tuple s in ps:
          if r[A] == s[A]:
            yield (r,s)
```

<u>Cost:</u>

$$\mathrm{P}(R) + \frac{P(R)}{B-1} P(S)$$

1. Load in B-1 pages of R at a time (leaving 1 page each free for S & output)

2. **For each (B-1)-page segment of R, load each page of S**

Note: Faster to iterate over the *smaller* relation first!

Block Nested Loop Join (BNLJ)

Given ***B+1*** pages of memory

```
Compute R ⋈ S on A:
  for each B-1 pages pr of R:
    for page ps of S:
      for each tuple r in pr:
        for each tuple s in ps:
          if r[A] == s[A]:
            yield (r,s)
```

Cost:

$$\mathrm{P}(R) + \frac{P(R)}{B-1}P(S)$$

1. Load in B-1 pages of R at a time (leaving 1 page each free for S & output)
2. For each (B-1)-page segment of R, load each page of S
3. **Check against the join conditions**

BNLJ can also handle non-equality constraints

Block Nested Loop Join (BNLJ)

Given ***B+1*** pages of memory

```
Compute R ⋈ S on A:
  for each B-1 pages pr of R:
    for page ps of S:
      for each tuple r in pr:
        for each tuple s in ps:
          if r[A] == s[A]:
            yield (r,s)
```

Again, ***OUT*** could be bigger than P(R)*P(S)… but usually not that bad

Cost:

$$\mathrm{P}(R) + \frac{P(R)}{B-1} P(S) + \mathrm{OUT}$$

1. Load in B-1 pages of R at a time (leaving 1 page each free for S & output)
2. For each (B-1)-page segment of R, load each page of S
3. Check against the join conditions
4. **Write out**

BNLJ vs. NLJ: Benefits of IO Aware

- In BNLJ, by loading larger chunks of R, we minimize the number of full *disk reads* of S
  - We only read all of S from disk for ***every (B-1)-page segment of R***!
  - Still the full cross-product, but more done only *in memory*

NLJ

P(R) + T(R)*P(S) + OUT

BNLJ

$P(R) + \frac{P(R)}{B-1} P(S)$ + OUT

BNLJ is faster by roughly $\frac{(B-1)T(R)}{P(R)}$!

BNLJ vs. NLJ: Benefits of IO Aware

- Example:
  - R: 500 pages
  - S: 1000 pages
  - 100 tuples / page
  - We have 12 pages of memory (B = 11)

Ignoring OUT here…

- NLJ: Cost = 500 + **50,000*1000** = **50 Million IOs ~= <u>140 hours</u>**

- BNLJ: Cost = 500 + $\frac{500*1000}{10}$ = **50 *Thousand* IOs ~= <u>0.14 hours</u>**

A very real difference from a small change in the algorithm!

Smarter than Cross-Products

Smarter than Cross-Products: From Quadratic to Nearly Linear

- All joins that compute the ***full cross-product*** have some **quadratic** term
  - For example we saw:

NLJ $P(R) + T(R)P(S) + OUT$

BNLJ $P(R) + \frac{P(R)}{B-1}P(S) + OUT$

- Now we'll see some (nearly) linear joins:
  - ~ O(P(R) + P(S) + ***OUT***), where again ***OUT*** could be quadratic but is usually better

We get this gain by ***taking advantage of structure***- moving to equality constraints ("equijoin") only!

Index Nested Loop Join (INLJ)

```
Compute R ⋈ S on A:
  Given index idx on S.A:
    for r in R:
      s in idx(r[A]):
        yield r,s
```

Cost:

P(R) + T(R)****L*** + OUT

where ***L*** is the IO cost to access all the distinct values in the index; assuming these fit on one page, $\mathrm{L} \sim 3$ is good est.

→ We can use an **index** (e.g. B+ Tree) to ***avoid doing the full cross-product!***