CS150A Database

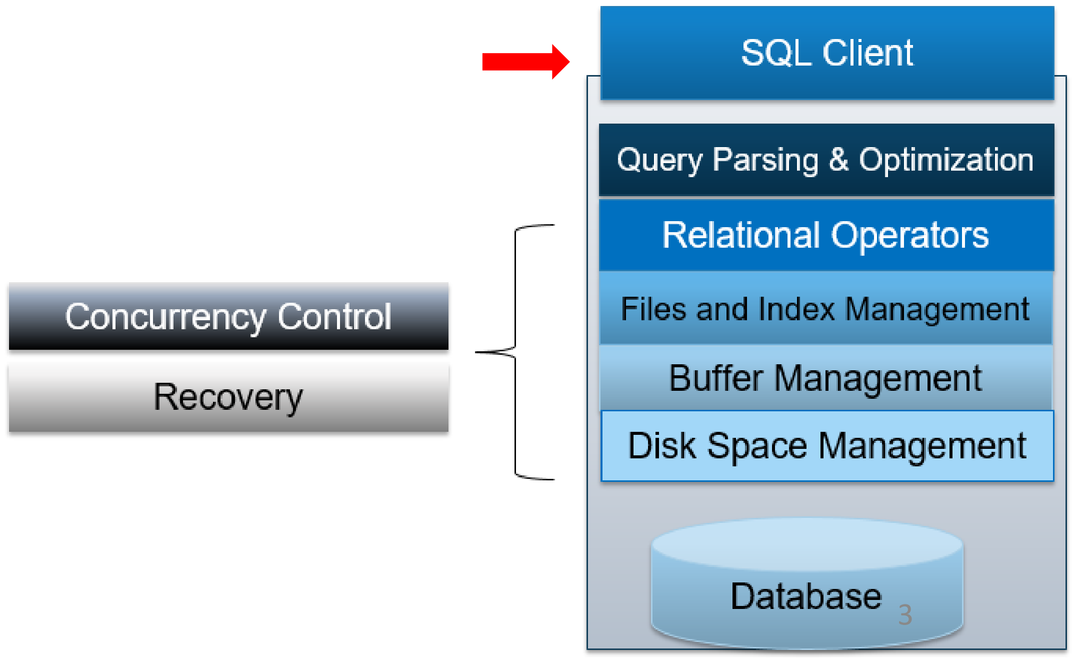
Midterm Course Review

Apr. 17, 2024

Database

1. SQL
2. Disk, Buffers and Files
3. Index and B+ Trees
   - refinement
4. Buffer Manager
5. Relational Algebra
6. Sorting and Hashing
7. Iterations and Joins
8. Query Optimization
9. Transactions and Concurrency
10. Recovery
11. ER Modeling & FD
12. Parallel Query Processing
13. Distributed Transactions
14. NoSQL
15. MapReduce & Spark
16. DM & ML
    - Data Warehouse / Lake
    - Machine Learning
      - *k*-means
      - Linear Regression

1. SQL


SQL Client
Query Parsing & Optimization
Relational Operators
Files and Index Management
Buffer Management
Disk Space Management
Database
Concurrency Control
Recovery

SQL DML:
Basic Single-Table Queries

- **SELECT** [**DISTINCT**] *<column expression list>*
  FROM *<single table>*
  [**WHERE** *<predicate>*]
  [**GROUP BY** *<column list>*
  [**HAVING** *<predicate>*]]
  [**ORDER BY** *<column list>*]
  [**LIMIT** <integer>];

Arithmetic Expressions

Combining Predicates

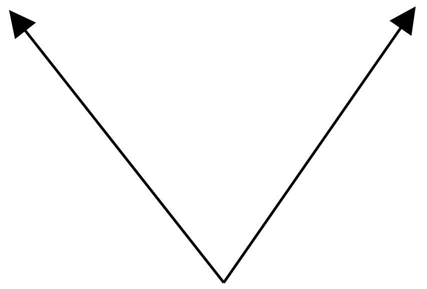
- Subtle connections between:
  - Boolean logic in WHERE (i.e., AND, OR)
  - Traditional Set operations (i.e. INTERSECT, UNION)
    - Set: a collection of distinct elements
    - Standard ways of manipulating/combining sets
      - Union
      - Intersect
      - Except
    - Treat tuples within a relation as elements of a set

Nested Queries

- *Names of sailors who' ve reserved boat #102:*

```
SELECT S.sname
FROM   Sailors S
WHERE  S.sid IN
    (SELECT  R.sid
    FROM    Reserves R
    WHERE   R.bid=102)
```

```
SELECT  S.sname
FROM  Sailors S
WHERE  EXISTS
    (SELECT  R.sid
    FROM  Reserves R
    WHERE  R.bid=102)
```



subquery

We've seen: IN, EXISTS

Can also have: NOT IN, NOT EXISTS

Other forms: op ANY, op ALL

Join Variants

SELECT <*column expression list*>
FROM *table_name*
  [**INNER** | **NATURAL**
   | {**LEFT** |**RIGHT** | **FULL** } {**OUTER**}] JOIN *table_name*
  ON <*qualification_list*>
WHERE …

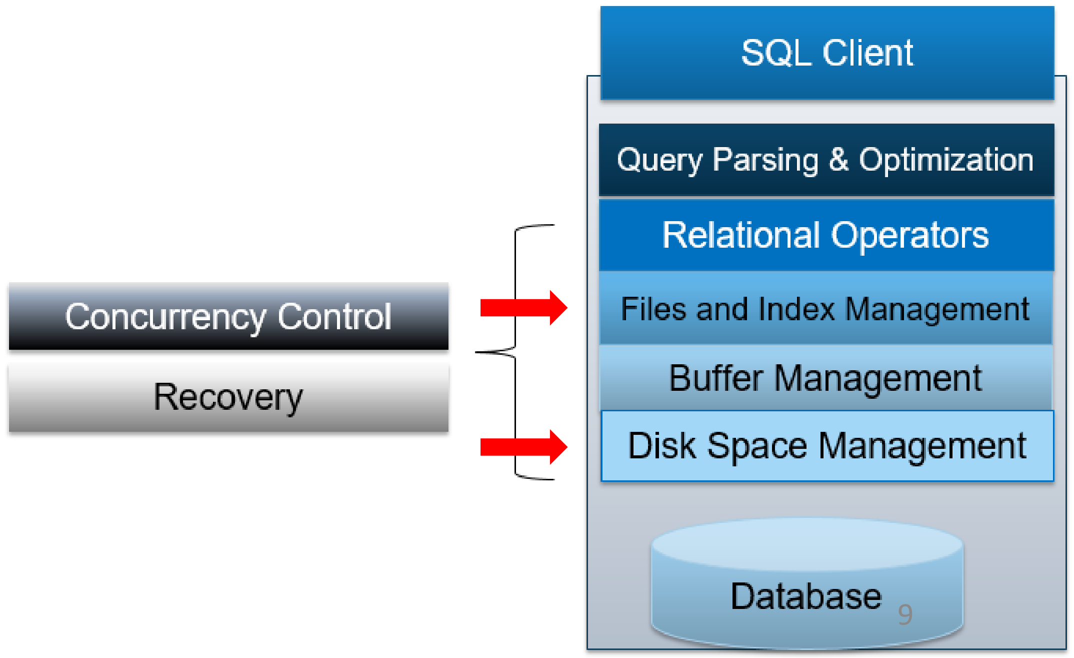
- INNER is default
- Inner join what we've learned so far
  - Same thing, just with different syntax.

Brief Detour: Null Values

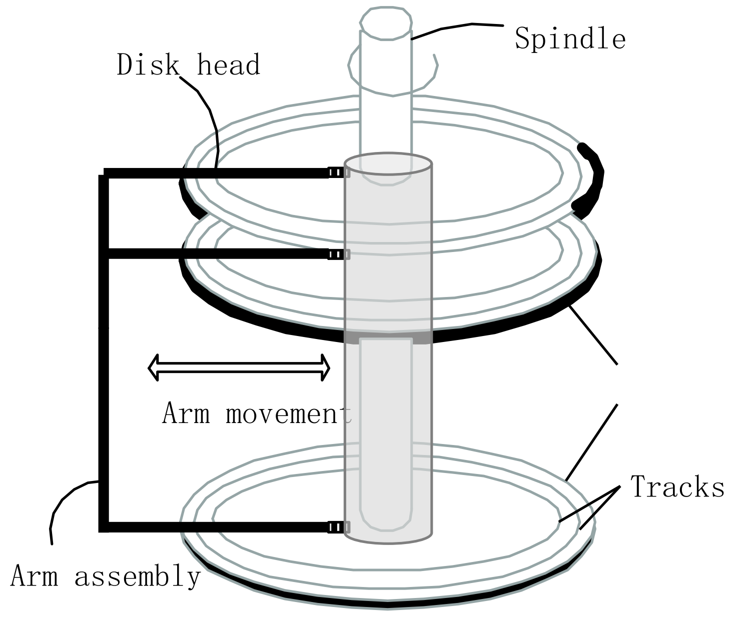
- Field values are sometimes unknown
  - SQL provides a special value NULL for such situations.
  - Every data type can be NULL
- The presence of null complicates many issues. E.g.:
  - Selection predicates (WHERE)
  - Aggregation
- But NULLs comes naturally from Outer joins

- NULL op NULL is NULL
- WHERE NULL: do not send to output
- Boolean connectives: 3-valued logic
- Aggregates ignore NULL-valued inputs

2. Disk, Files and Buffers



Arranging Blocks on Disk



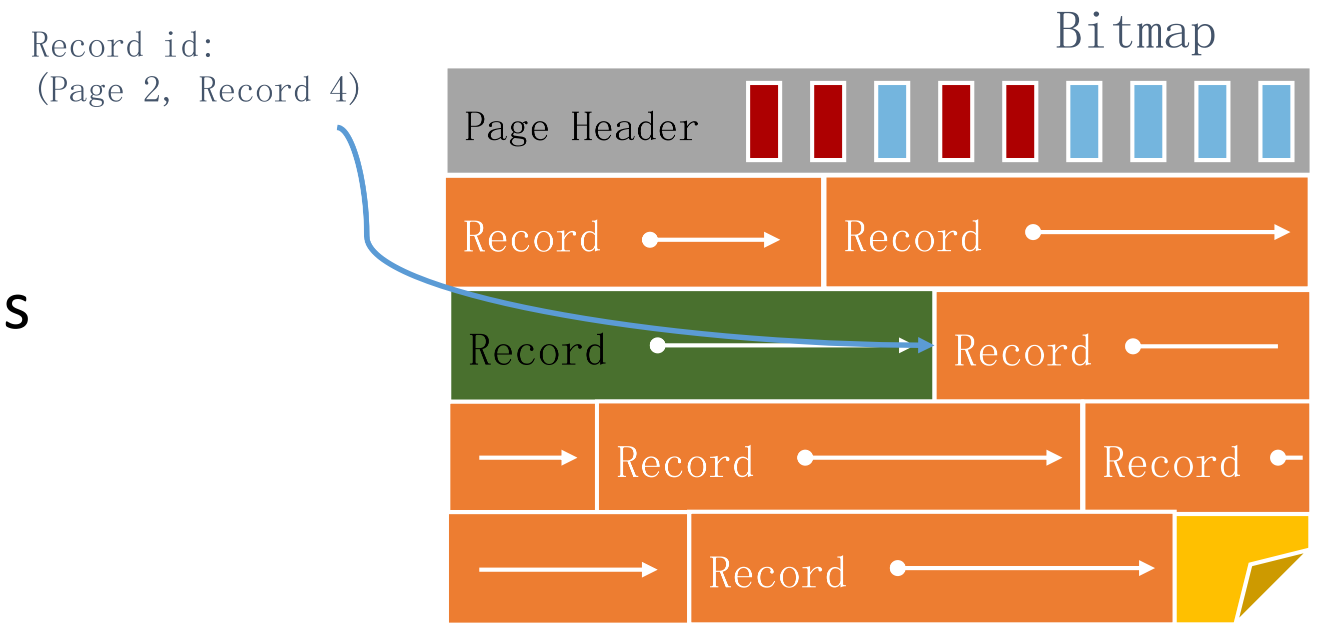
- **'Next'** block concept:
    - sequential blocks on same track, followed by
    - blocks on same cylinder, followed by
    - blocks on adjacent cylinder
- Arrange file pages sequentially by 'next' on disk
    - minimize seek and rotational delay.
- For a **sequential scan**, *pre-fetch*
    - several blocks at a time!
- **Read large consecutive blocks**

Key to lower I/O cost:
reduce seek time and rotational delays

Random vs. sequential disk access (10x)

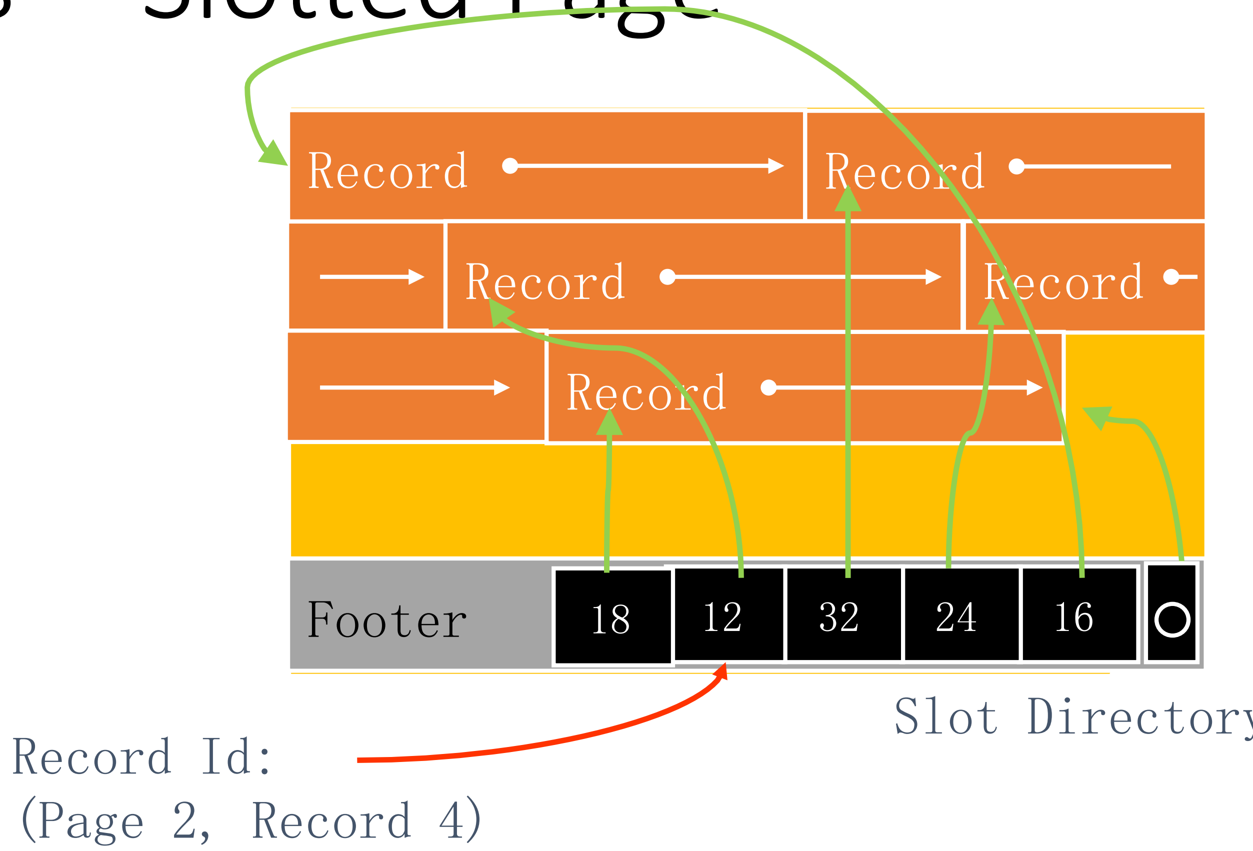
Fixed Length Records: Unpacked

- Bitmap denotes “slots” with records
- Record id: record number in page
- **Insert**: find first empty slot
- **Delete**: Clear bit



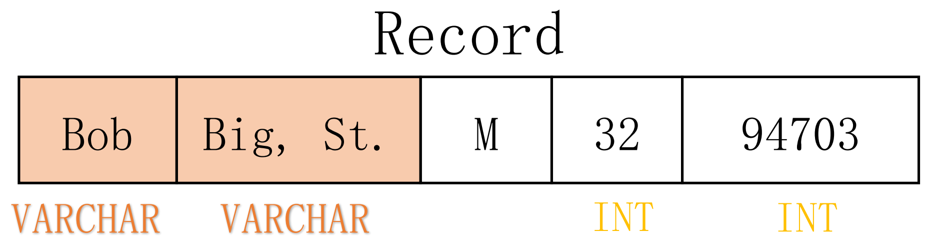
Variable Length Records -- Slotted Page

- Introduce slot directory in footer
  - Pointer to free space
  - Length + Pointer to beginning of record
    - reverse order
- Record ID = location in slot table
  - from right
- Delete?
  - e.g., 4th record on the page

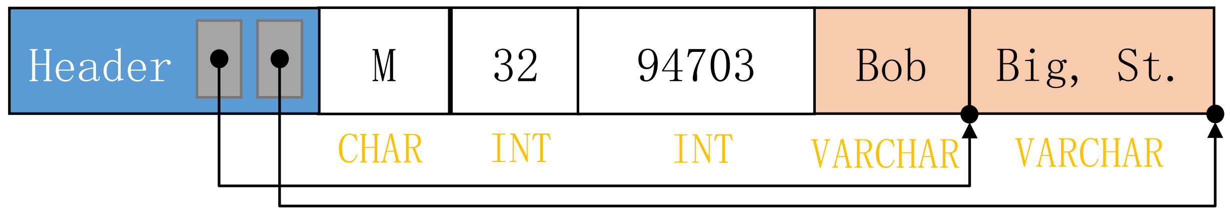


Record Formats: Variable Length

- What happens if fields are variable length?



- Introduce a record header



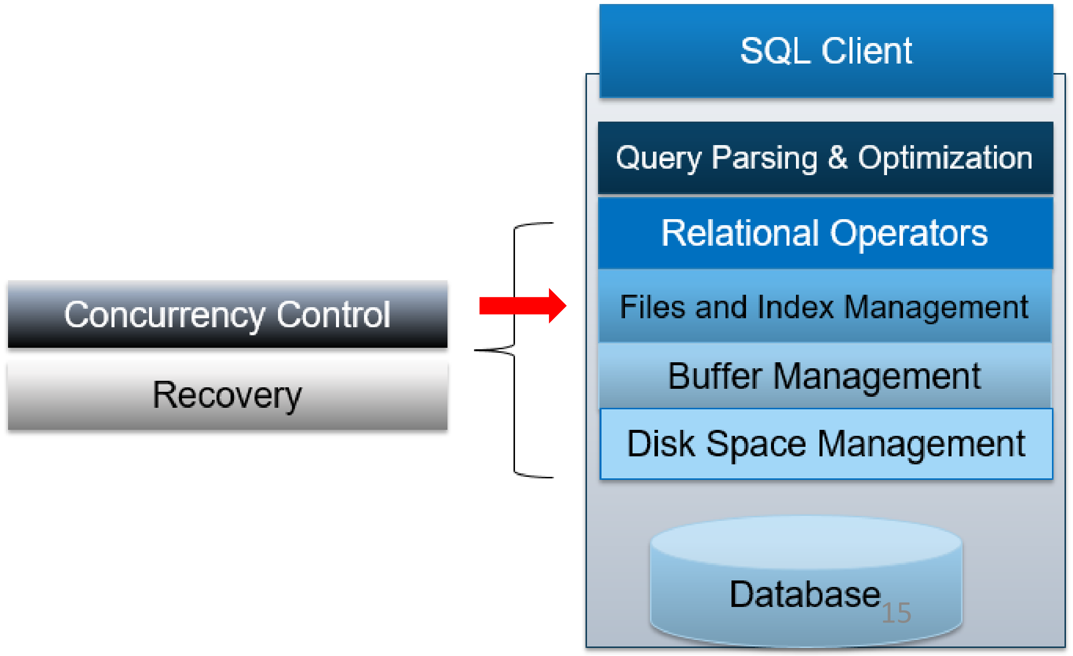
- Direct access & no "escaping", other advantages?
  - Handle null fields easily →
  - useful for fixed length records too!

Cost of Operations Complete

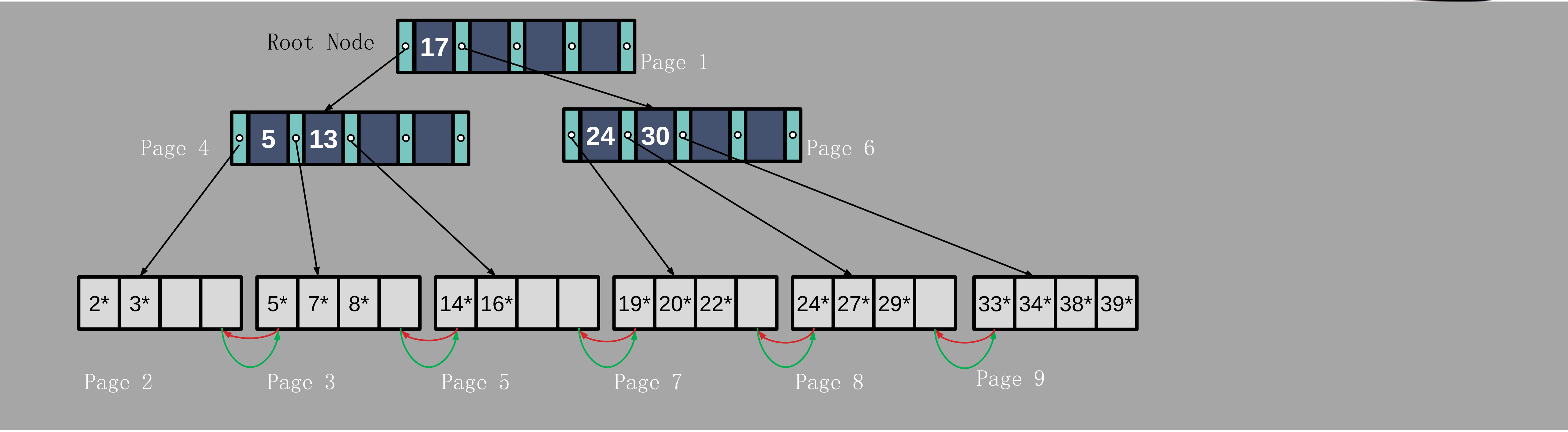
	Heap File	Sorted File
Scan all records	B*D	B*D
Equality Search	0.5*B*D	$(\log_2 B)*D$
Range Search	B*D	$((\log_2 B)+pages)*D$
Insert	2*D	$((\log_2 B)+B)*D$
Delete	(0.5*B+1)*D	$((\log_2 B)+B)*D$

- **B:** The number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block
- Can we do better?
  - Indexes!

3. Index and B+ Trees



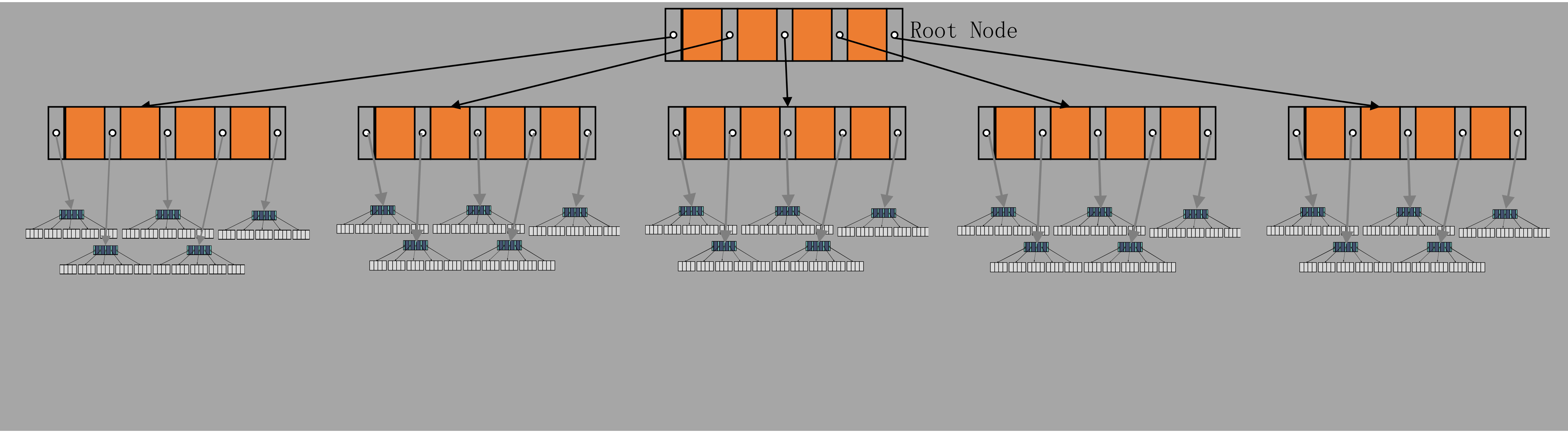
Example of a B+ Tree



- Occupancy Invariant
  - Each interior node is at least partially full:
    - d <= #entries <= 2d
    - d: order of the tree (max fan-out = 2d + 1)
- Data pages at bottom need not be stored in logical order
  - Next and prev pointers

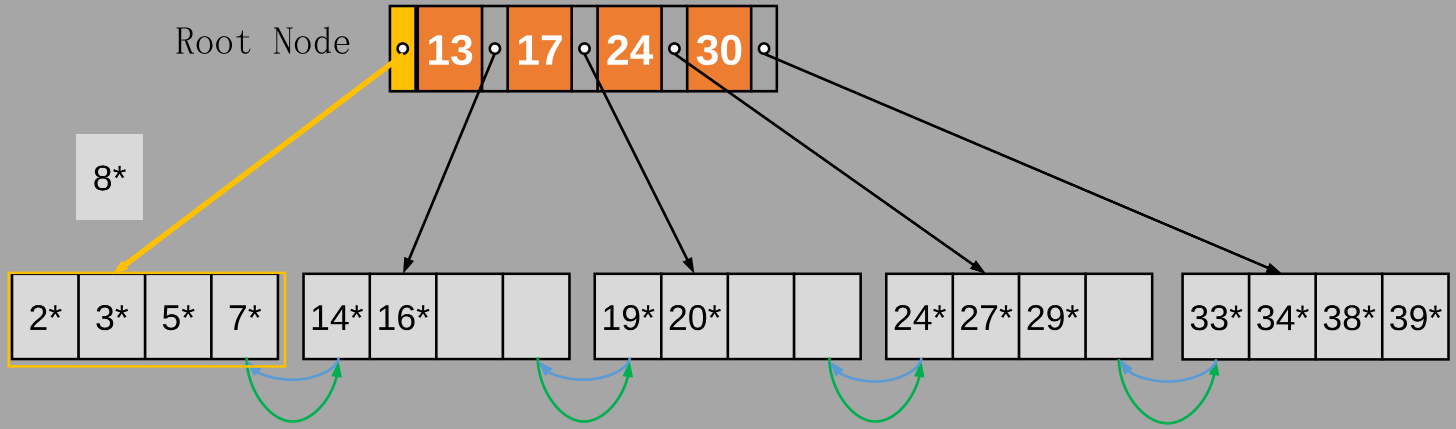
What is the value of d?
2
What about the root?
The root is special
Why not in sequential order?
Data pages allocated dynamically

B+ Trees and Scale



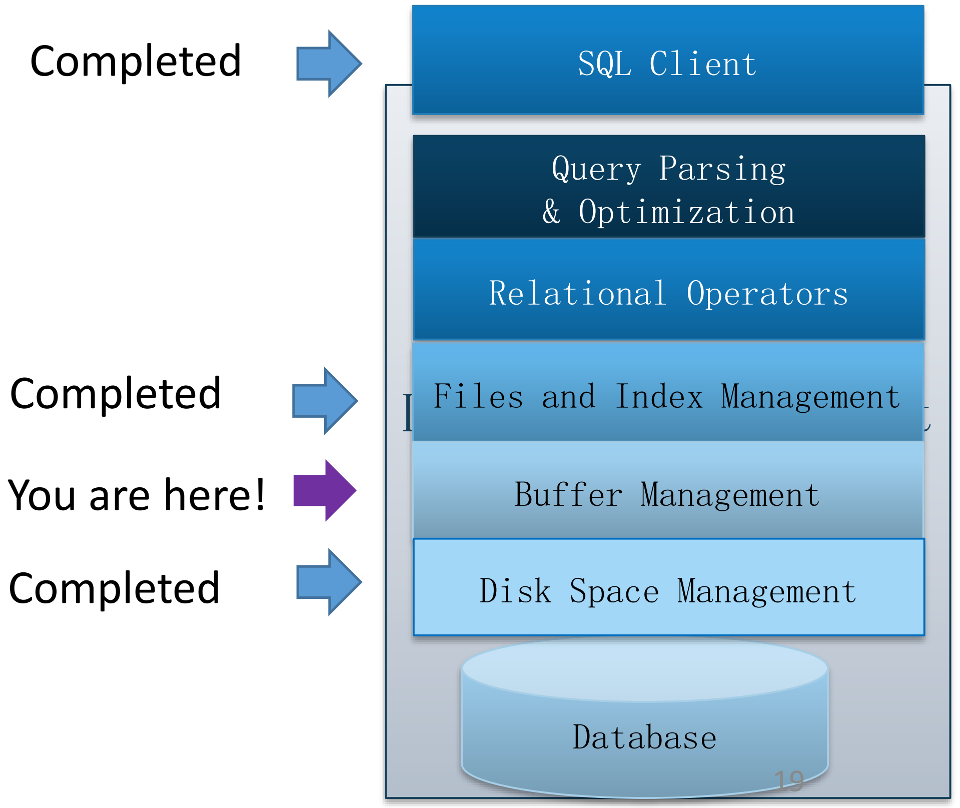
- How big is a height 3 B+ tree
  - d = 2 → Fan-out?
  - Fan-out = 2d + 1 = 5
  - **Height 3:** 5^3 x 4= 500 Records

Inserting 8* into a B+ Tree: Insert



- Find the correct leaf
    - Split leaf if there is not enough room

4. Buffer Manager



Answers to Our Previous Questions

1. **Handling dirty pages**
   - How will the buffer manager find out?
     - Dirty bit on page
   - What to do with a dirty page?
     - Write back via disk manager

2. **Page Replacement**
   - How will the buffer mgr know if a page is "in use"?
     - **Page pin count**
   - If buffer manager is full, which page should be replaced?
     - **Page replacement policy**
       - Least-recently-used (LRU), Clock
       - Most-recently-used (MRU)

FrameId	PageId	Dirty?	Pin Count
1	1	N	0
2	2	Y	1
3	3	N	0
4	6	N	2
5	4	N	0
6	5	N	0

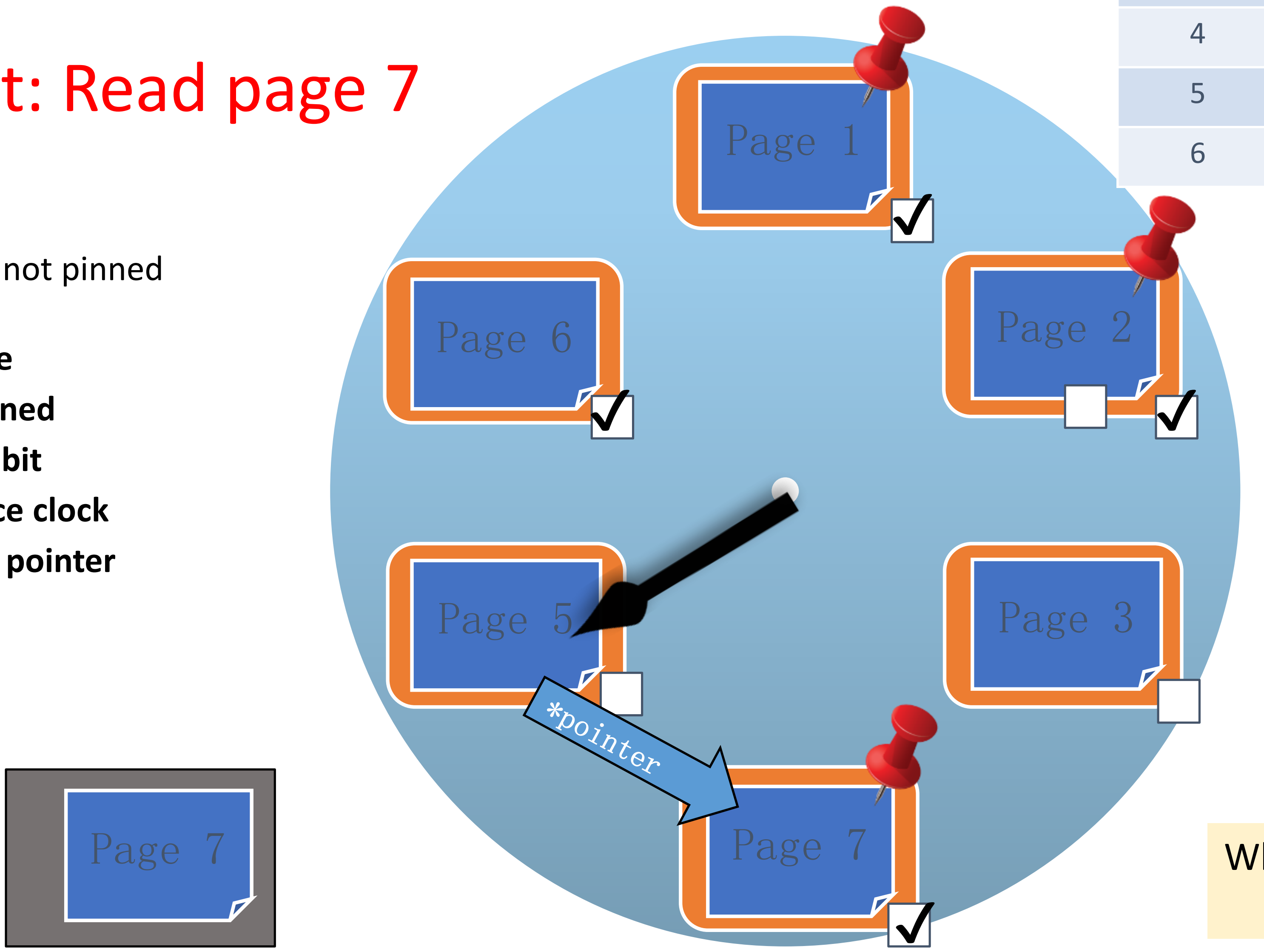
Keep an in-memory index (hash table) on PageId

Clock Policy State: Illustrated

Request: Read page 7

Current frame not pinned
Ref bit unset:

- **Replace**
- **Set pinned**
- **Set ref bit**
- **Advance clock**
- **Return pointer**



FrameId	PageId	Dirty?	Pin Count	Ref Bit
1	1	N	1	1
2	2	N	1	1
3	3	N	0	1
4	4	N	0	0
5	5	N	0	0
6	6	N	0	1

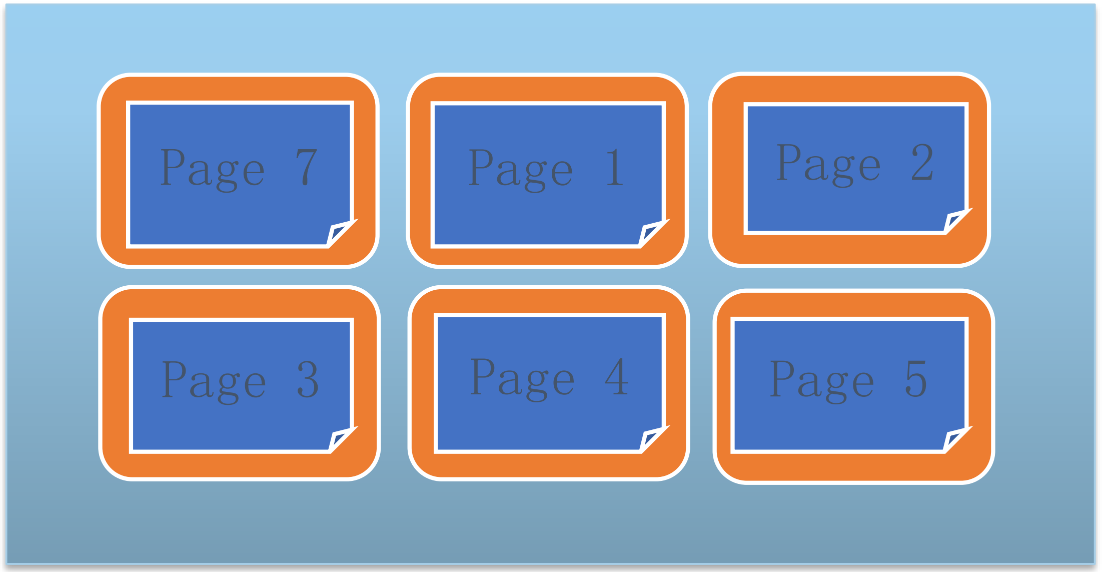
Clock Hand
1

When might they perform poorly?

- repeated scans of big files

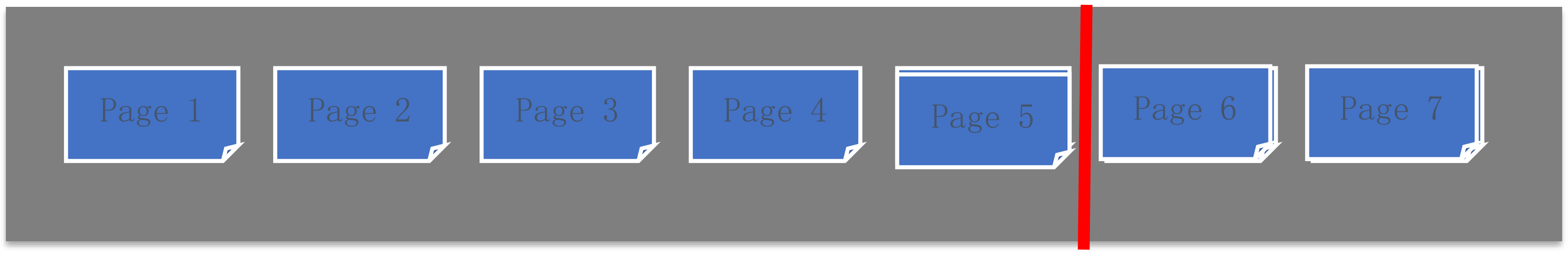
Repeated Scan (LRU): Read Page 5

- Cache Hits: 0
- Attempts: 12



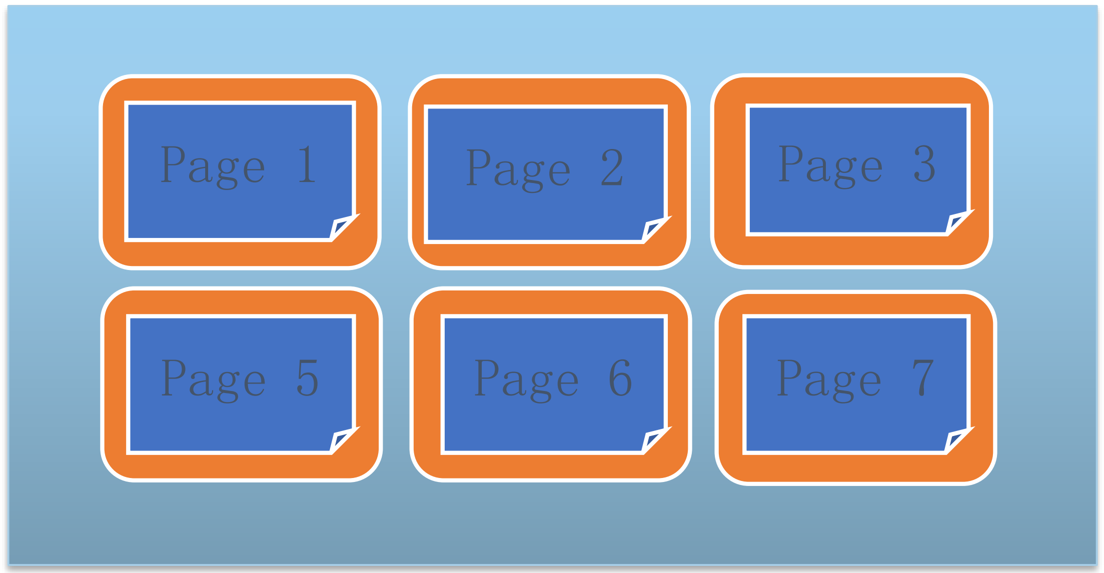
Get the picture? A worst-case scenario!
"Sequential Flooding"

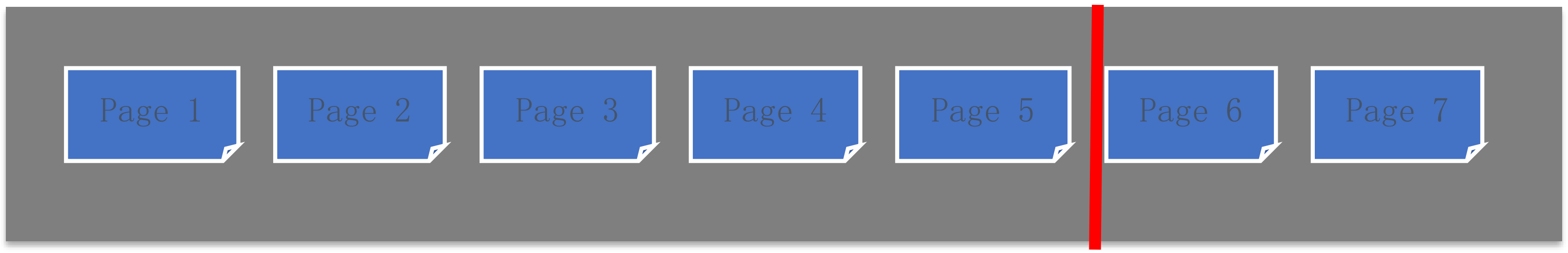
Disk Space Manager



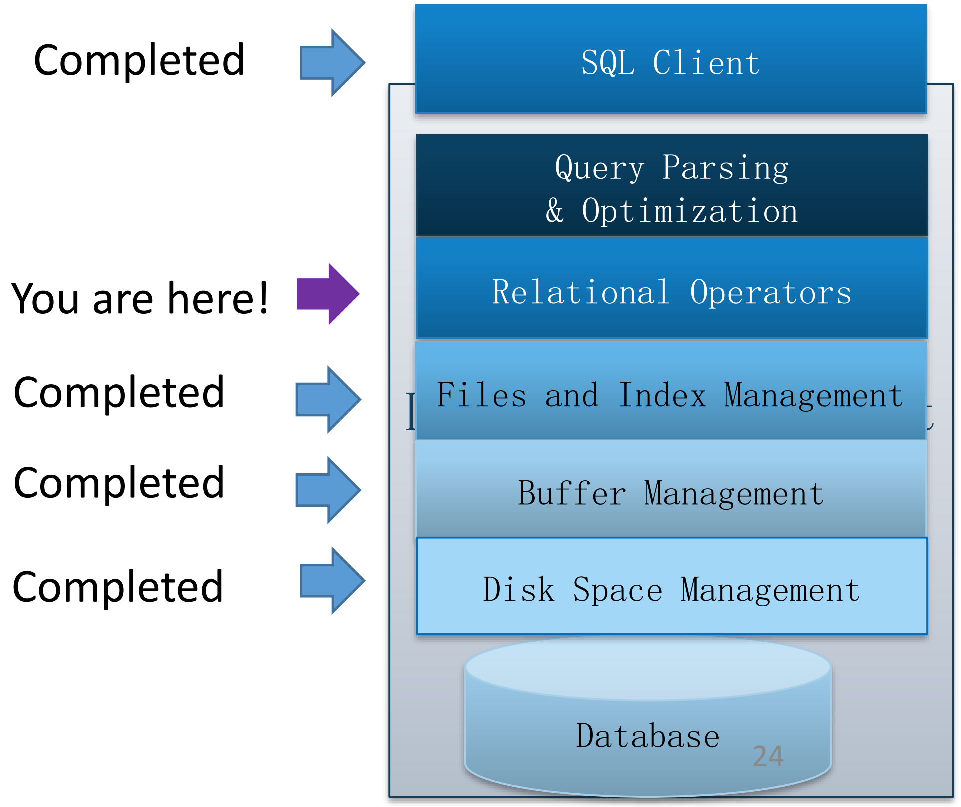
Repeated Scan (MRU): Read Page 5

- Cache Hits: 10
- Attempts: 19





5. Relational Algebra



* SQL vs Relational Algebra

SQL Query

```
SELECT S.name
FROM Reserves R, Sailors S
WHERE R.sid = S.sid
AND R.bid = 100
AND S.rating > 5
```

Query Parser
& Optimizer

Relational Algebra

$$\pi_{S.name}(\sigma_{bid=100 \land rating>5}(\text{Reserves} \bowtie_{R.sid=S.sid} \text{Sailors}))$$

Equivalent to…

(Lo

On-the-fly

tor

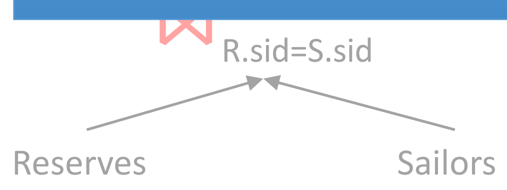
SQL

A **declarative** expression of the query result

But actually wi
produce…

Relational Algebra

Operational description of a computation.



Operator Code

B+-Tr
Indexed
Iterator

Systems execute relational algebra query plan.

Relational Algebra Preliminaries

- Algebra of operators on relation instances
- $\pi_{\text{S.name}}(\sigma_{\text{R.bid=100} \wedge \text{S.rating>5}}(\text{R} \bowtie_{\text{R.sid=S.sid}} \text{S}))$
  - Closed: result is also a relation instance
    - Enables rich composition!
  - Typed: input schema determines output
    - Can statically check whether queries are legal.
  - Pure relational algebra has set semantics
    - No duplicate tuples in a relation instance
    - vs. SQL, which has multiset (bag) semantics

Relational Algebra Operators

- Unary Operators: on **single relation**
  - **Projection** (π): Retains only desired columns (vertical)
  - **Selection** (σ): Selects a subset of rows (horizontal)
  - **Renaming** (ρ): Rename attributes and relations.
- Binary Operators: on **pairs of relations**
  - **Union** ($\cup$): Tuples in r1 or in r2.
  - **Set-difference** (—): Tuples in r1, but not in r2.
  - **Cross-product** ($\times$): Allows us to combine two relations.
- Compound Operators: common "*macros*" for the above
  - **Intersection** ($\cap$): Tuples in r1 and in r2.
  - **Joins** ($\bowtie_{\theta}$, $\bowtie$): Combine relations that satisfy predicates

6. Sorting and Hashing

General External Merge Sort

- More than 3 buffer pages. How can we utilize them?
  - Big batches in pass 0, many streams in merge passes

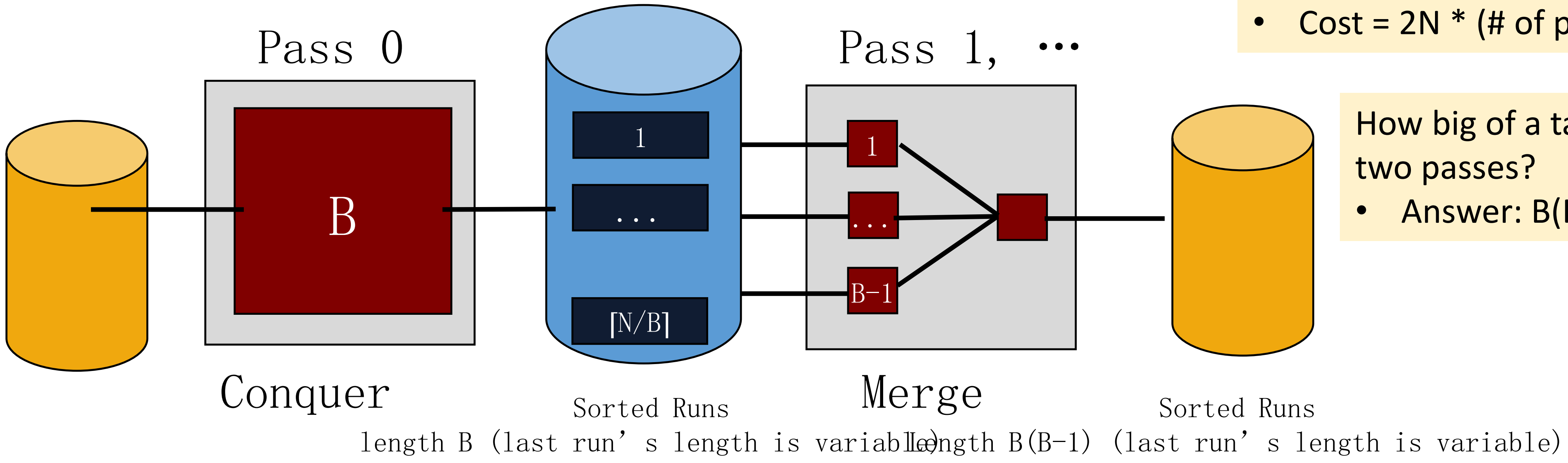
- To sort a file with N pages using B buffer pages:
  - Pass 0: use B buffer pages. Produce $\lceil N/B \rceil$ sorted runs of B pages each.
  - Pass 1, 2, …, etc.: merge B-1 runs at a time.

Cost

- Number of passes: $1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$
- Cost = 2N * (# of passes)

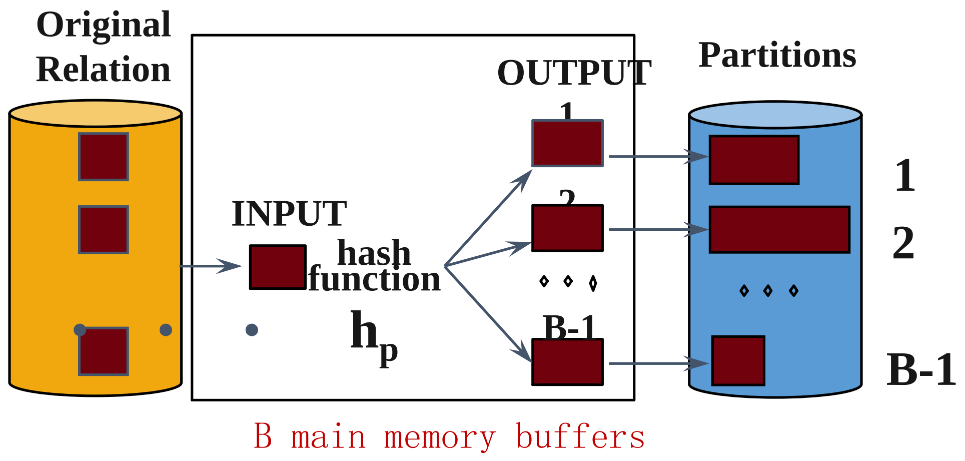
How big of a table can we sort in two passes?

- Answer: B(B-1).

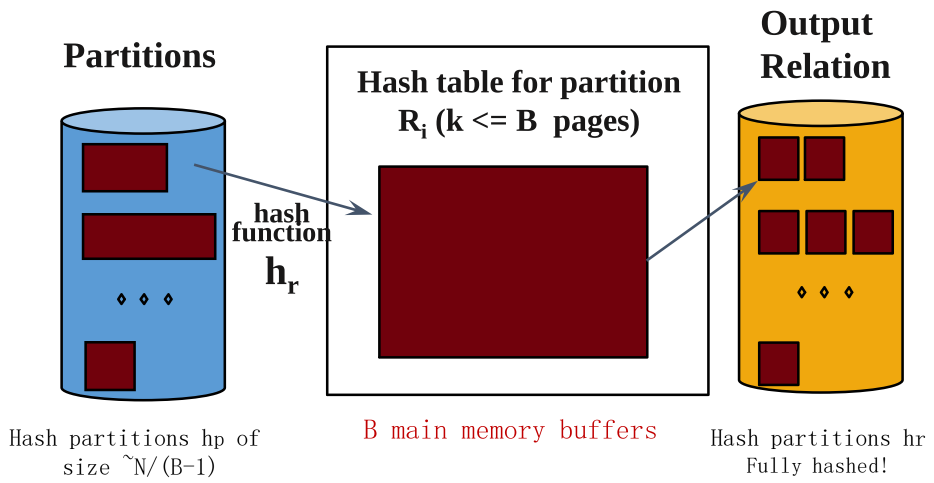

Pass 0
B
Conquer
1
. . .
⌈N/B⌉
Sorted Runs
length B (last run' s length is variable)
Pass 1, …
1
. . .
B−1
Merge
Sorted Runs
length B(B−1) (last run' s length is variable)

Two Phases of Hash

- Partition:
  (Divide)



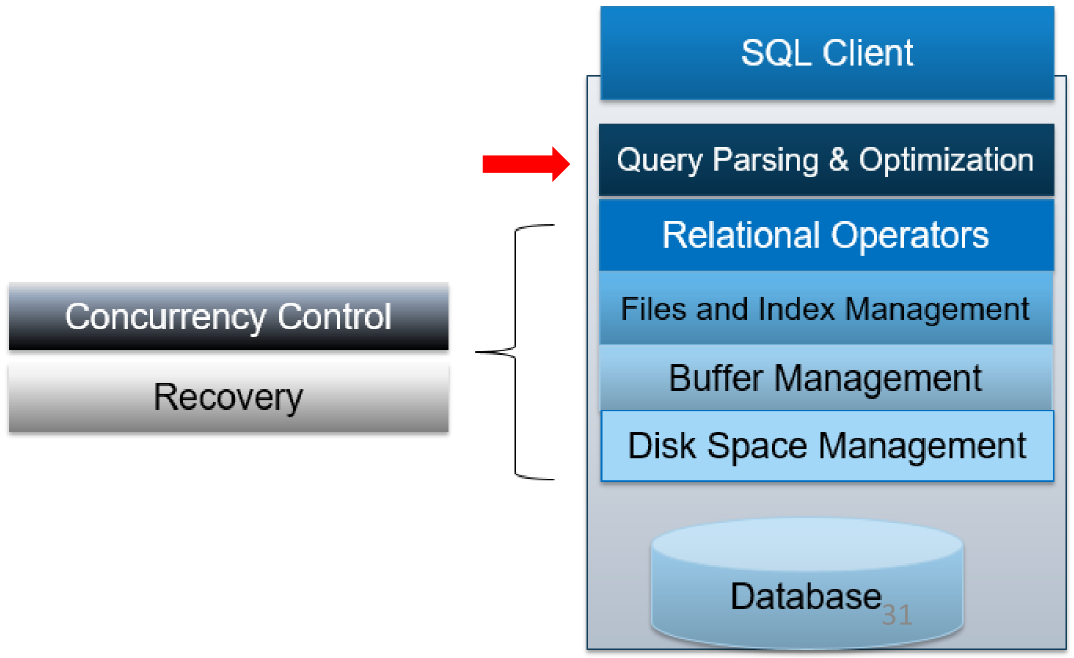
- Rehash:
  (Conquer)



cost = 2*N*(#passes) = 4*N IO's
(includes initial read, final write)

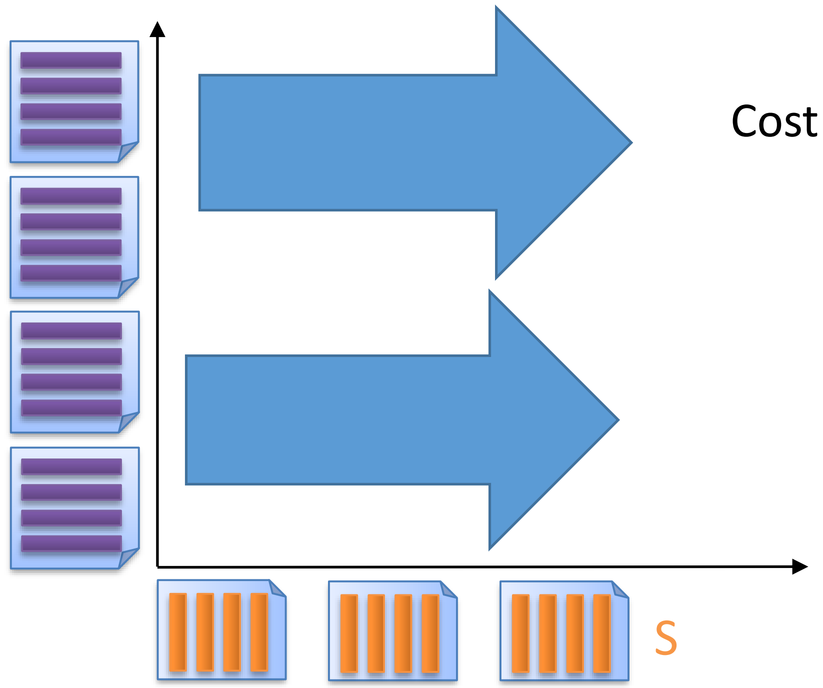
How big of a table can we sort in two passes?
- Answer: B(B-1).

7. Iterations and Joins



“Chunk” ~~“Block”~~ Nested Loop Join

```
for each rchunk of B-2 pages of R:
    for each spage of S:
        for all matching tuples in spage and rchunk:
            add <rtuple, stuple> to result buffer
```

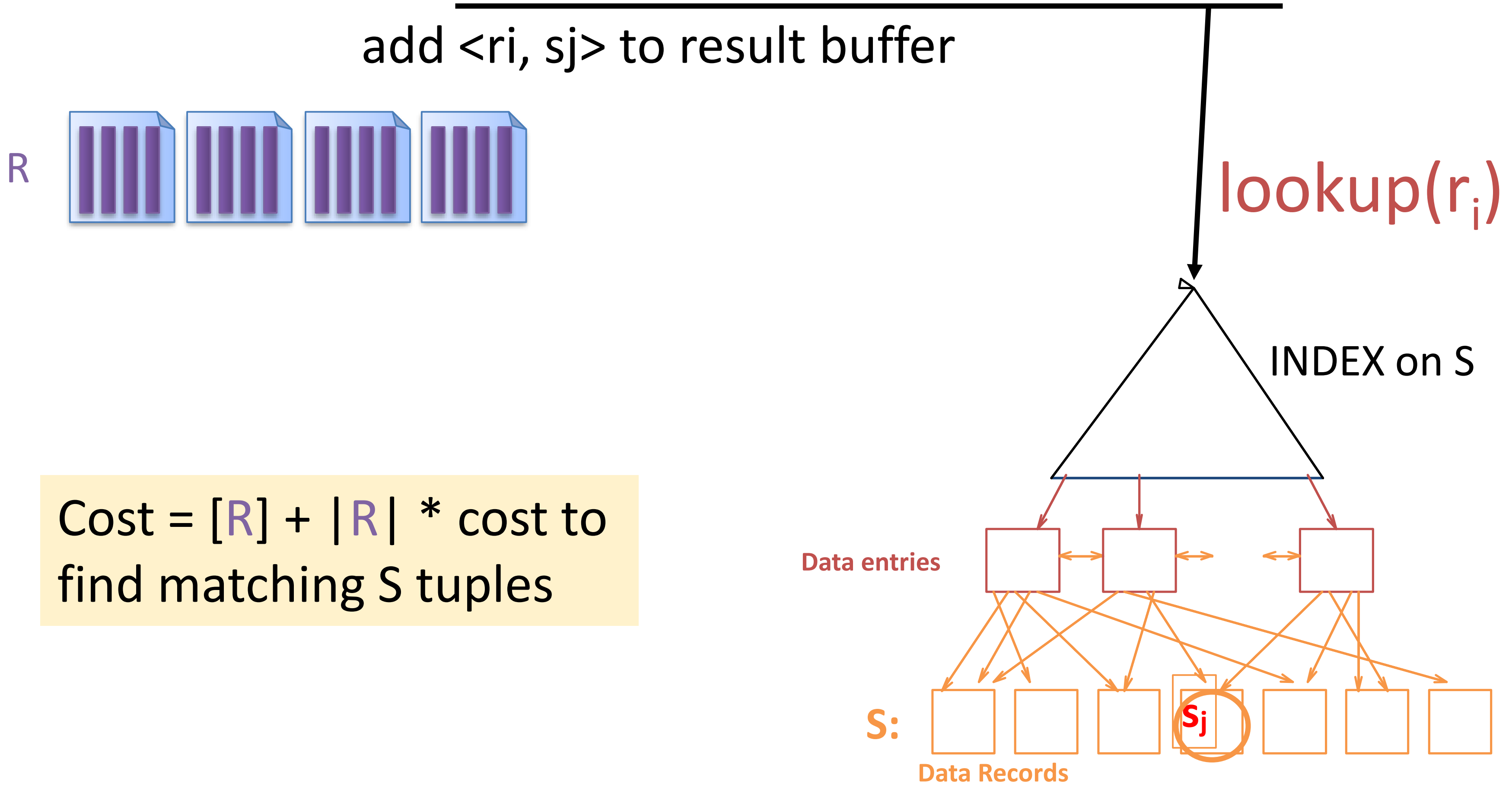


Cost = [R] + ⌈ [R]/(B-2) ⌉ * [S]
= 1000 + ⌈ 1000/(B-2) ⌉ * 500
= 6,000 for B=102 (~100x better than Page NL!)

Which is the inner / outer relation?

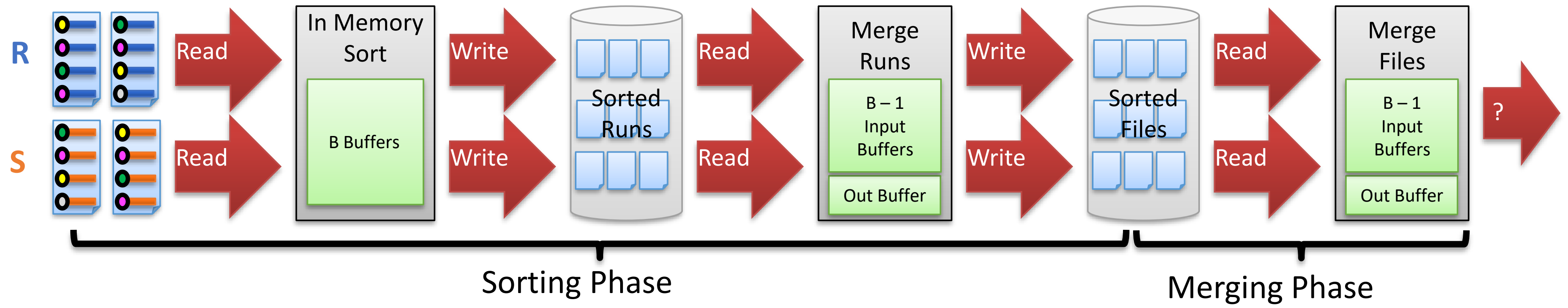
Index Nested Loops Join

```
foreach tuple r in R do
    foreach tuple s in S where ri == sj do
        add <ri, sj> to result buffer
```



Cost = [R] + |R| * cost to find matching S tuples

Cost of Sort-Merge Join

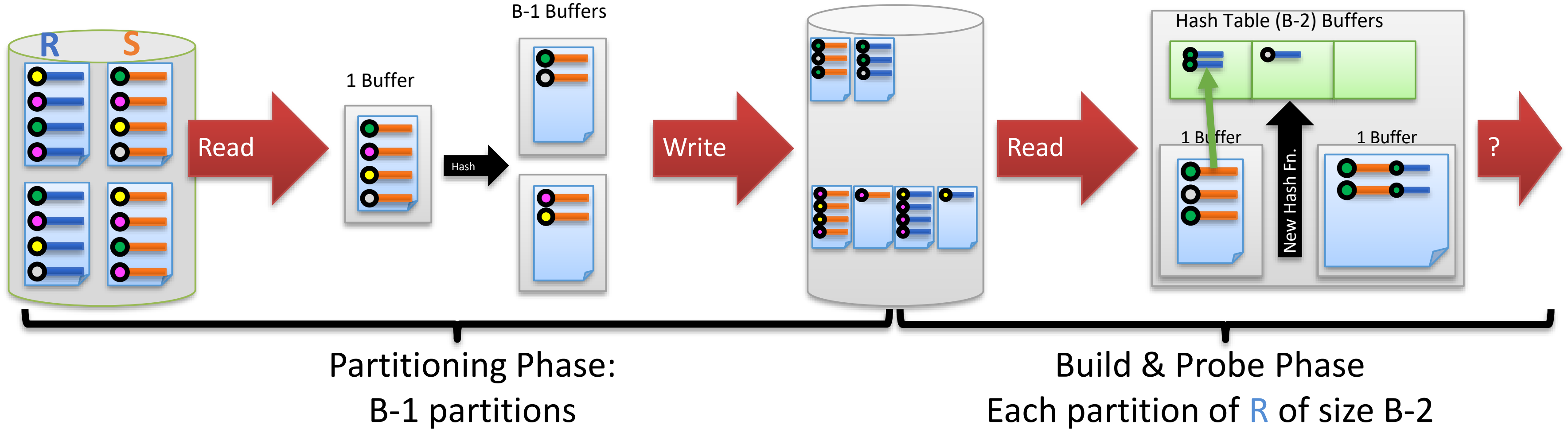


- Cost: Sort R + Sort S + ([R]+[S])
  - But in worst case, last term could be |R| *[S] (very unlikely!)
  - Q: what is worst case?

- Question: How big does the buffer have to be to sort both R and S in two passes each?

- Suppose buffer B > $\sqrt{(\max([R], [S]))}$
  - Both R and S can be sorted in 2 passes
  - 4*1000 + 4*500 + (1000 + 500) = 7500

Summary of Grace Hash Join

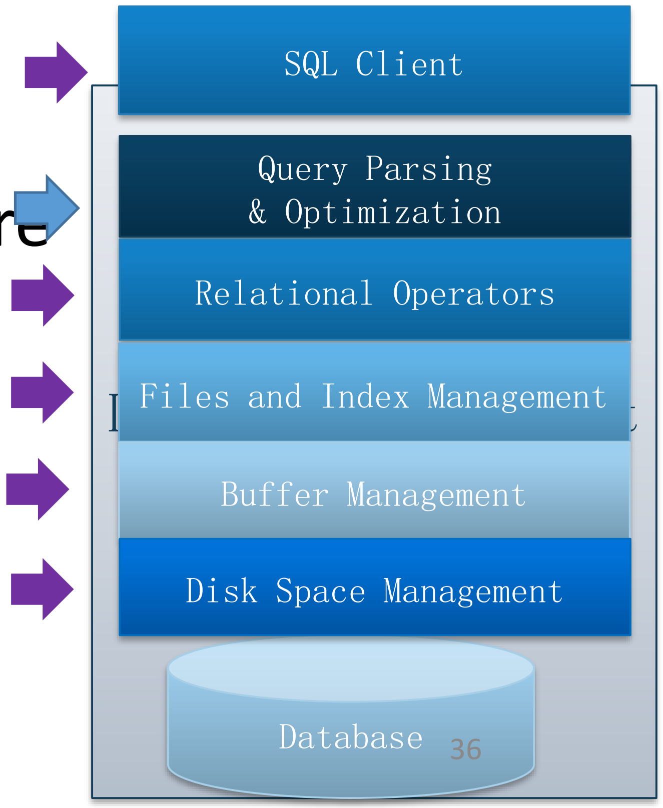


- Partitioning phase: read+write both relations
  - ® 2([R]+[S]) I/Os
- Matching phase: read both relations, forward output
  - ® [R]+[S]
- Total cost of 2-pass hash join = 3([R]+[S])

- **Memory Requirements?**
- Build hash table on R with uniform partitioning
  - **Partitioning Phase divides R into (B-1) runs of size [R] / (B-1)**
  - **Matching Phase requires each [R] / (B-1) < (B-2)**
  - **R < (B-1) (B-2) ≈ B^2**
- Note: no constraint on size of S (probing relation)!

8. Query Optimization

- Completed
- You are here
- Completed
- Completed
- Completed
- Completed



Big Picture of System R Optimizer

- Works well for up to 10-15 joins.
- **Plan Space**: Too large, must be pruned.
  - Algorithmic insight:
    - Many plans could have the same "overpriced" subtree
    - Ignore all those plans
  - Common heuristic: consider only left-deep plans
  - Common heuristic: avoid Cartesian products
- Cost estimation
  - Very inexact, but works ok in practice.
  - Stats in system catalogs used to estimate sizes & costs
  - Considers combination of CPU and I/O costs.
  - System R's scheme has been improved since that time.
- Search Algorithm: Dynamic Programming

Result Size Estimation

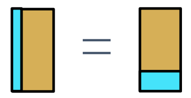
- Result cardinality = Max # tuples * **product** of all selectivities.
- Term col=value (given Nkeys(I) on col)
  - sel = 1/NKeys(I)
- Term col1=col2 (handy for joins too…)
  - sel = 1/MAX(NKeys(I1), NKeys(I2))
  - Why MAX? See bunnies in 2 slides…
- Term col>value
  - sel = (High(I)-value)/(High(I)-Low(I) + 1)
- Note, if missing the needed stats, assume 1/10!!!

selectivity = |output| / |input|

Upshot

- Know how to compute selectivities for basic predicates
  - The original Selinger version
  - The histogram version

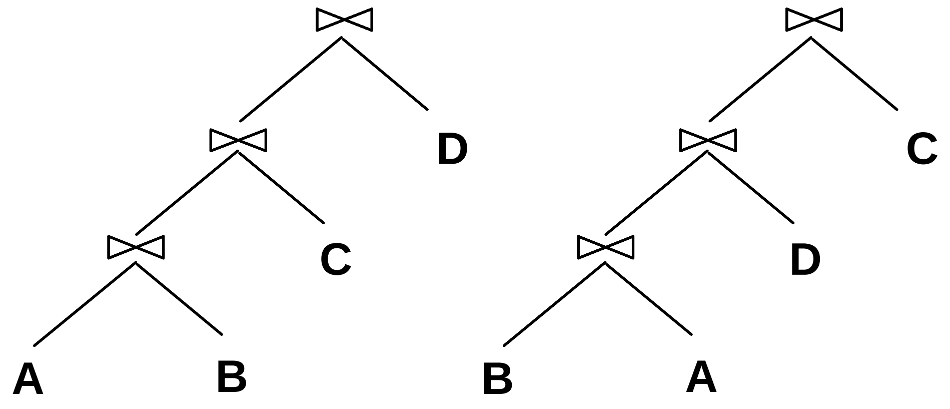
- **Assumption 1**: uniform distribution within histogram bins
  - Within a bin, fraction of range = fraction of count



- **Assumption 2**: independent predicates
  - Selectivity of AND = product of selectivities of predicates
  - Selectivity of OR = sum of selectivities of predicates - product of selectivities of predicates
  - Selectivity of NOT = 1 – selectivity of predicates

- Joins are not a special case
  - Simply compute the selectivity of all predicates
  - And multiply by the product of the table sizes

Enumeration of Left-Deep Plans



- Left-deep plans differ in
  - the order of relations
  - the access method for each leaf operator
  - the join method for each join operator

- Enumerated using N passes (if N relations joined):
  - **Pass 1:** Find best 1-relation plan for each relation
  - **Pass i:** Find best way to join result of an (i -1)-relation plan (as outer) to the i'th relation. (i between 2 and N.)

- For each subset of relations, retain only:
  - Cheapest plan overall, plus
  - Cheapest plan for each *interesting order* of the tuples.