Final Exam Review

Kathleen Durant
CS 3200 Northeastern University
Lecture 22

Outline for today

- Identify topics for the final exam
- Discuss format of the final exam
  - What will be provided for you and what you can bring (and not bring)
- Review content

Final Exam

- April 19, 2013 8:00 AM Shillman Hall
- Open books and open notes
  - But no portable devices (no laptops, no phones, etc.)
- 2 hour time period

Lectures for the final exam

- 9 lectures – all presentations are numbered with the corresponding lecture number
- **All lectures included**

Text chapters for the final exam

- Chapters 8-11
  8. Overview of storage and indexing
  9. Storing data: disks and files
  10. Tree-structured indexing
      - Including section on B+ trees in Chapter 17 (17.5.2)
  11. Hash-based indexing
- Chapters 12-15
  12. Query Evaluation
  13. External Sorting
  14. Evaluating Relational Operators
  15. Typical Relational Operator

Topics for the final exam

Topics

- File storage mechanisms
  - Abstraction:collection of records
  - Formats
  - Heap-based, Sorted, Indexed
  - RAID
- Buffer management
  - In relationship to the data manager
- Indexes
  - Primary vs. Secondary
  - Clustered vs. Unclustered
  - Tree-structured: ISAM, B+ trees
  - Hash-based indexes
- External Sort
- Query Evaluation
- Query Optimization
- NO SQL

Algorithms

- Cost model
  - Given a query, the approximate number of I/O's for different file storage mechanisms
- B+ tree bulk load
- Insertion/Deletion of records
  - B+ tree
  - ISAM
  - Extendible hashing
  - Linear hashing
- Query plan selection

Format of the final exam

- 1-2 Algorithmic/Calculation problems (40%)
  - I/O calculations
  - B+ tree insertion/deletion
  - Construct or Choose a query plan
- 1-2 open-ended responses (30%)
  - SQL vs. NO SQL
    - ACID vs. BASE
    - CAP theorem
  - Comparison of Join algorithms
  - Sort algorithms
- Some close-ended responses (30%)
  - Short collection of True and False
  - Multiple choice
  - Short definitions

Study Steps

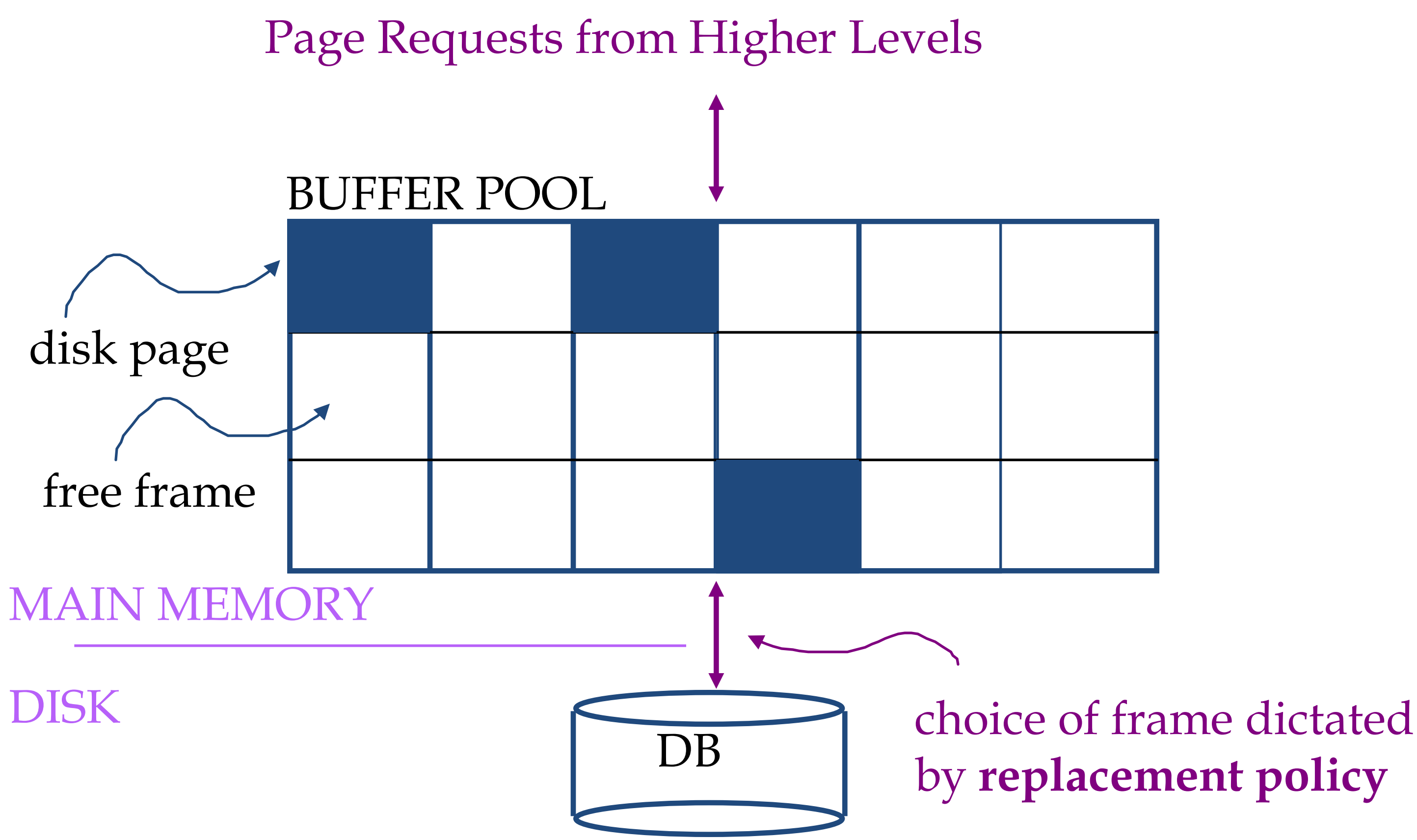
- Go over the lecture notes
- Read the book
  - Summary section of the chapters are written well
- Go over homework 3
- Ask questions in piazza or via email
- Organize a study sheet
- Review algorithms

CONTENT REVIEW

Disk Space Manager

- Lowest layer of DBMS software manages space on disk.
- Higher levels call upon this layer to:
  - allocate/de-allocate a page
  - read/write a page
- Request for a *sequence* of pages must be satisfied by allocating the pages sequentially on disk! Higher levels don't need to know how this is done, or how free space is managed.

Buffer Management in a DBMS

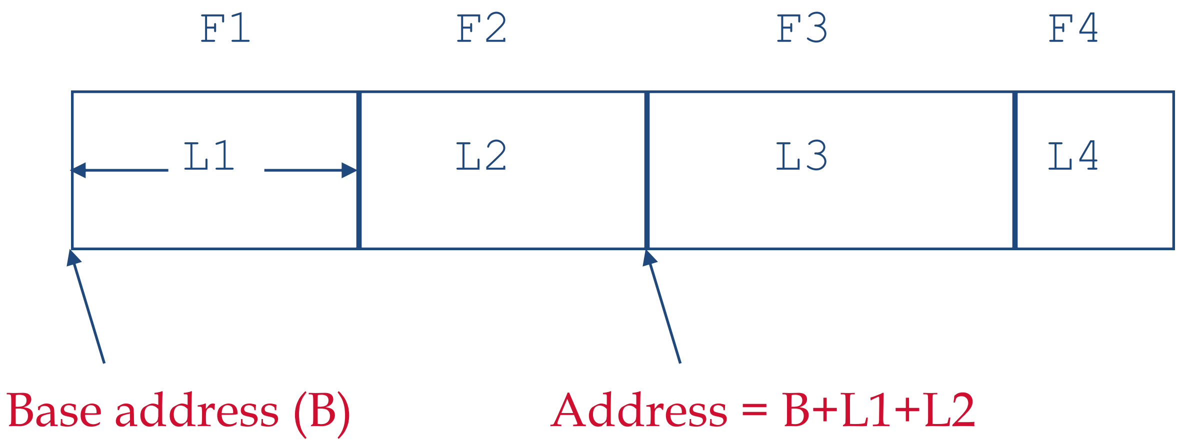


- *Data must be in RAM for DBMS to operate on it!*
- *Table of <frame#, pageid> pairs is maintained.*

File structure types

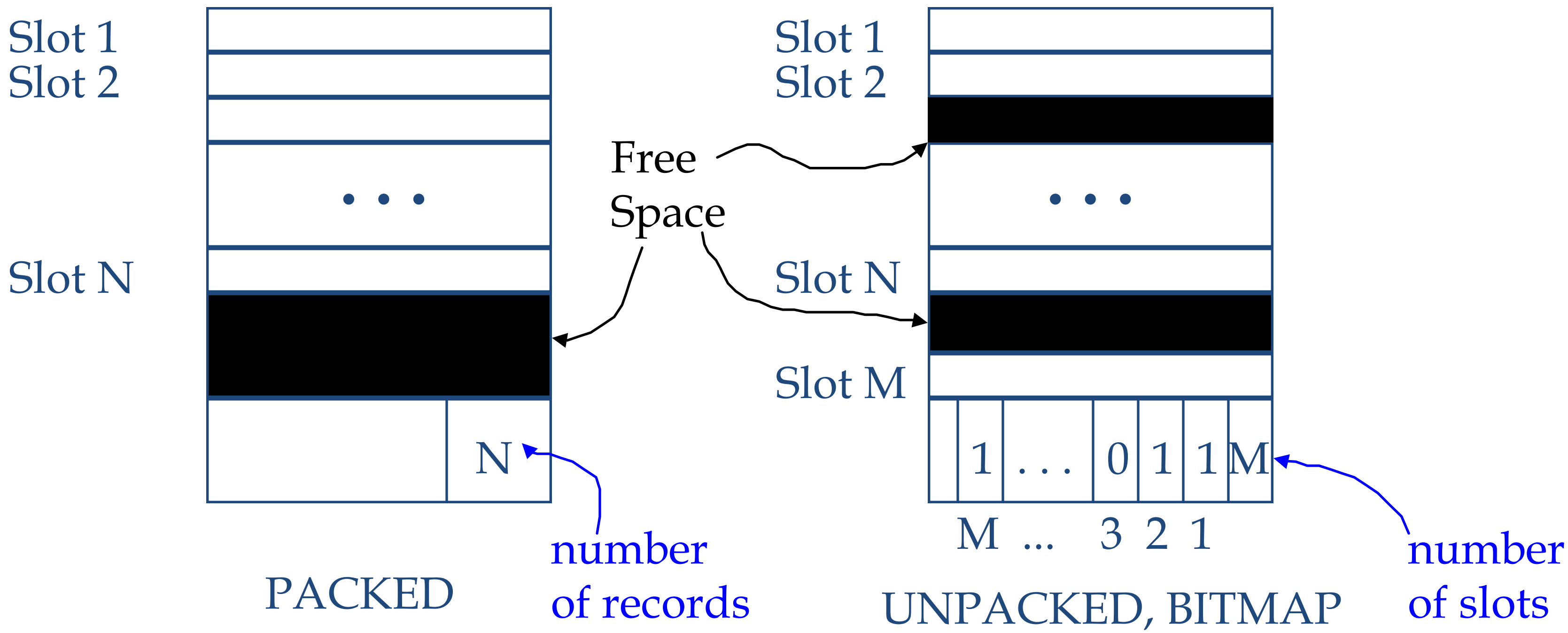
- Heap (random order) files
  - Suitable when typical access is a file scan retrieving all records.
- Sorted Files
  - Best if records must be retrieved in some order, or only a `range’ of records is needed.
- Indexes = data structures to organize records via trees or hashing.
  - Like sorted files, they speed up searches for a subset of records, based on values in certain (“search key”) fields
  - Updates are much faster than in sorted files.

Record Formats: Fixed Length



- Information about field types same for all records in a file; stored in *system catalogs.*
- Finding *i'th* field requires scan of record.

Page Formats: Fixed Length Records

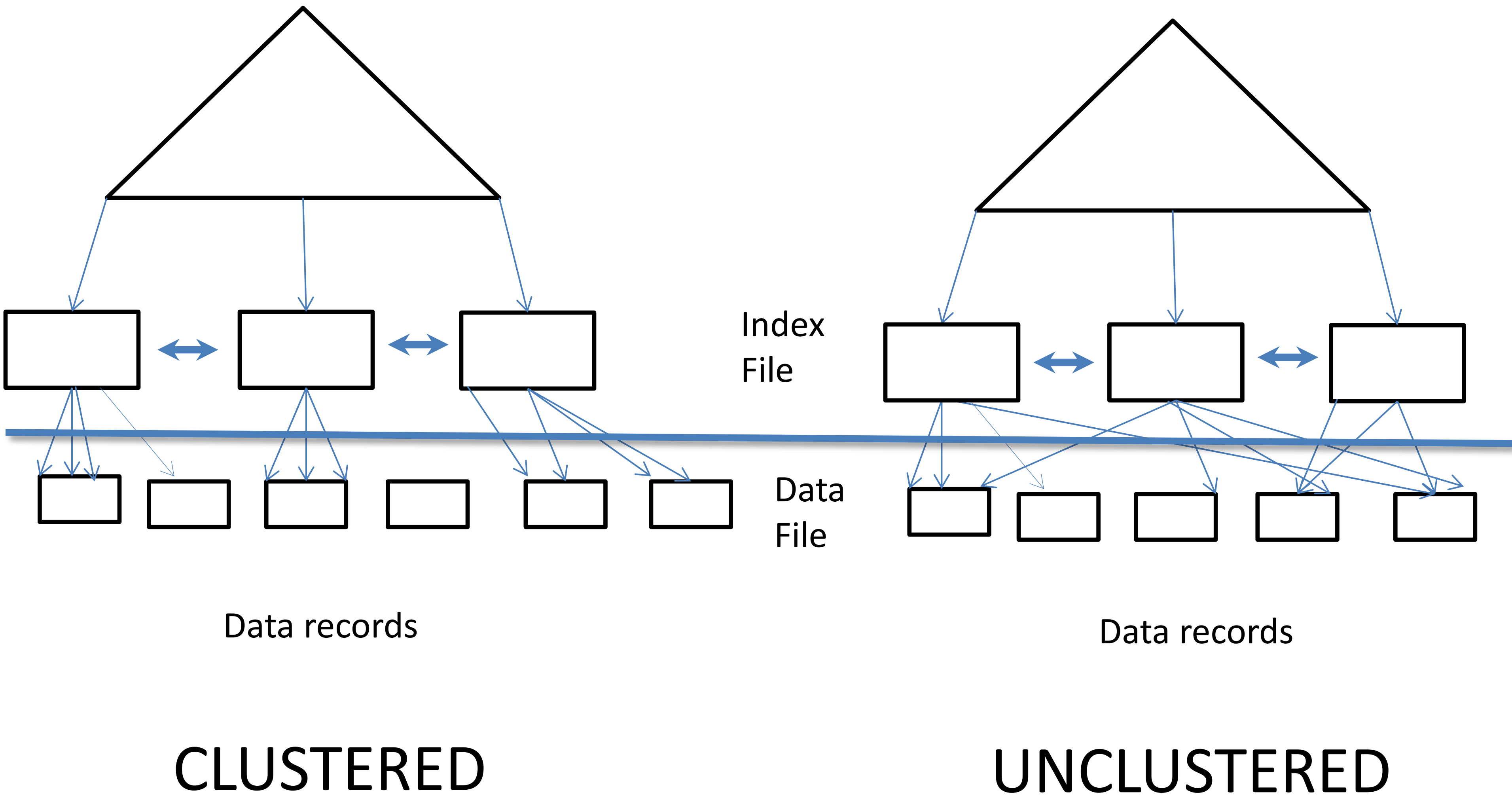


<u>Record id</u> = <page id, slot #>. In first alternative, moving records for free space management changes rid; may not be acceptable.

Index classification

- Primary vs. secondary: If search key contains primary key, then called primary index.
  - Unique index: Search key contains a candidate key.
- Clustered vs. unclustered: If order of data records is the same as, or `close to’, order of data entries, then called clustered index.
  - A file can be clustered on at most one search key.
  - Cost of retrieving data records through index varies greatly based on whether index is clustered or not.

Clustered vs. Unclustered Index



Cost Model Analysis

- We ignore CPU costs, for simplicity:
  - B: The number of data pages (Blocks)
  - R: Number of records per page (Records)
  - D: (Average) time to read or write a single disk page
- Measuring number of page I/O's
  - ignores gains of pre-fetching a sequence of pages; thus, even I/O cost is only approximated
- Average-case analysis; based on several simplifying assumptions
- Operations to measure
  - Scan whole table
  - Equality search
  - Range selection
  - Insert a record
  - Delete a record

Summary of workload

File Type	Scan	Equality Search	Range Search	Insert	Delete
Heap	BD	.5BD	BD	2D	Search + D
Sorted	BD	D $\log_2 B$	D$\log_2 B$ + # matching p.	Search + BD	Search + BD
Clustered	1.5BD	D $\text{Log}_F 1.5B$	D$\text{Log}_F 1.5B$ + **# matched pages**	Search + D	Search + D
Unclustered tree index	BD(R + 0.15)	D(1+ $\log_F 0.15B$)	D($\text{Log}_F 0.15B$ + **# matching records**)	D(3 + $\log_F 0.15B$)	Search + 2D
Unclustered Hash index	BD(R + 0.125)	2D	BD	4D	Searches + 2D

RAID Goals

- Disk Array: Arrangement of several disks that gives abstraction of a single, large disk
- Goals: Increase performance and reliability.
  - high capacity and high speed by using multiple disks in parallel
  - high reliability by storing data redundantly, so that data can be recovered even if a disk fails
- Two main techniques:
  - Data striping: Data is partitioned; size of a partition is called the striping unit. Partitions are distributed over several disks.
  - Redundancy: More disks -> more failures. Redundant information allows reconstruction of data if a disk fails.

Levels of Raid

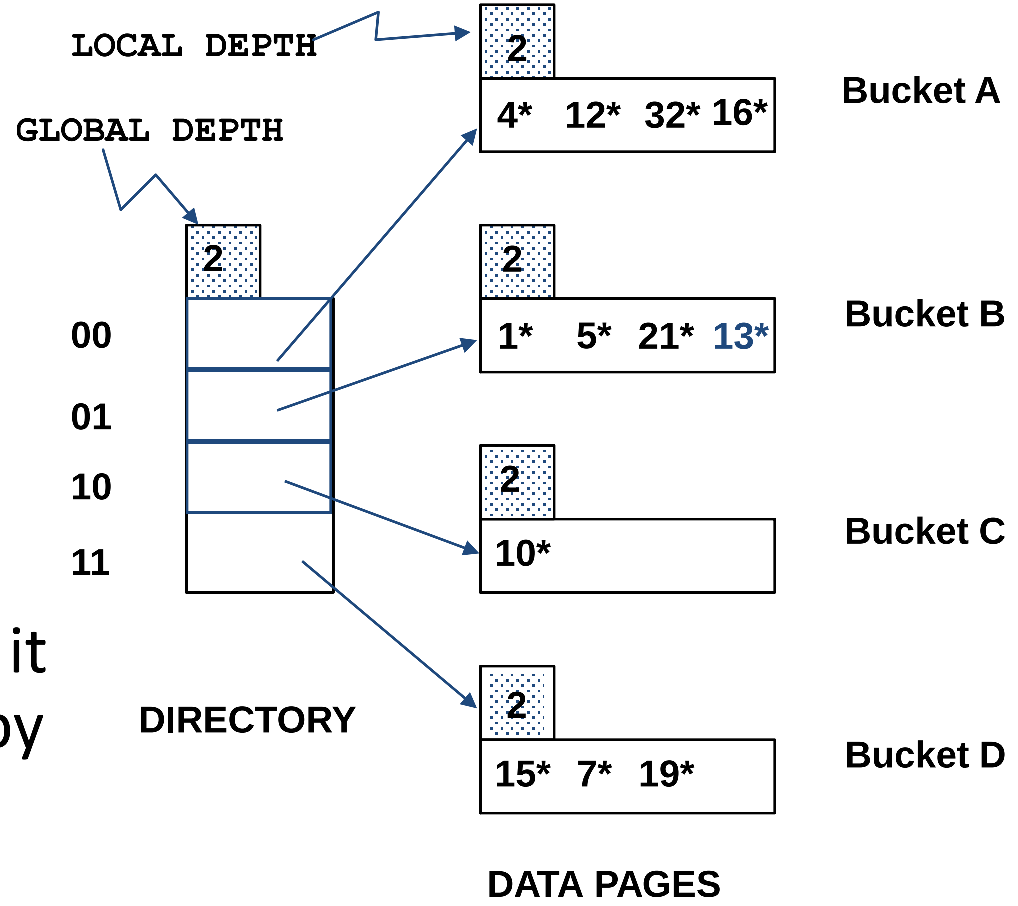
- **RAID Level 0**: Block striping; non-redundant.
  - Used in high-performance applications where data lost is not critical.
- **RAID Level 1**: Mirrored disks with block striping
  - Offers best write performance.
  - Popular for applications such as **storing log files in a database system.**
- **RAID Level 2**: Memory-Style Error-Correcting-Codes (ECC) with bit striping.
- **RAID Level 3**: Bit-Interleaved Parity
  - a single parity bit is enough for error correction, not just detection
    - When writing data, corresponding parity bits must also be computed and written to a parity bit disk
    - To recover data in a damaged disk, compute XOR of bits from other disks (including parity bit disk)
- **RAID Level 4: Block-Interleaved Parity;** uses block-level striping, and keeps a parity block on a separate disk for corresponding blocks from N other disks.
- **RAID Level 5:** Block-Interleaved Distributed Parity; partitions data and parity among all $N + 1$ disks, rather than storing data in N disks and parity in 1 disk.
- **RAID Level 6**: P+Q Redundancy scheme; similar to Level 5, but stores extra redundant information to guard against multiple disk failures

INDEXES

Extendible Hashing Algorithm

- Situation: Hash Bucket (primary page) becomes full. Why not re-organize file by *doubling* # of buckets?
  - Reading and writing all pages is expensive!
  - *Idea*: Use *directory of pointers to buckets,* double # of buckets by *doubling the directory,* splitting just the bucket that overflowed!
  - Directory much smaller than file, so doubling it is much cheaper. Only one page of data entries is split. *No overflow page*!
  - Trick lies in how hash function is adjusted!

Example

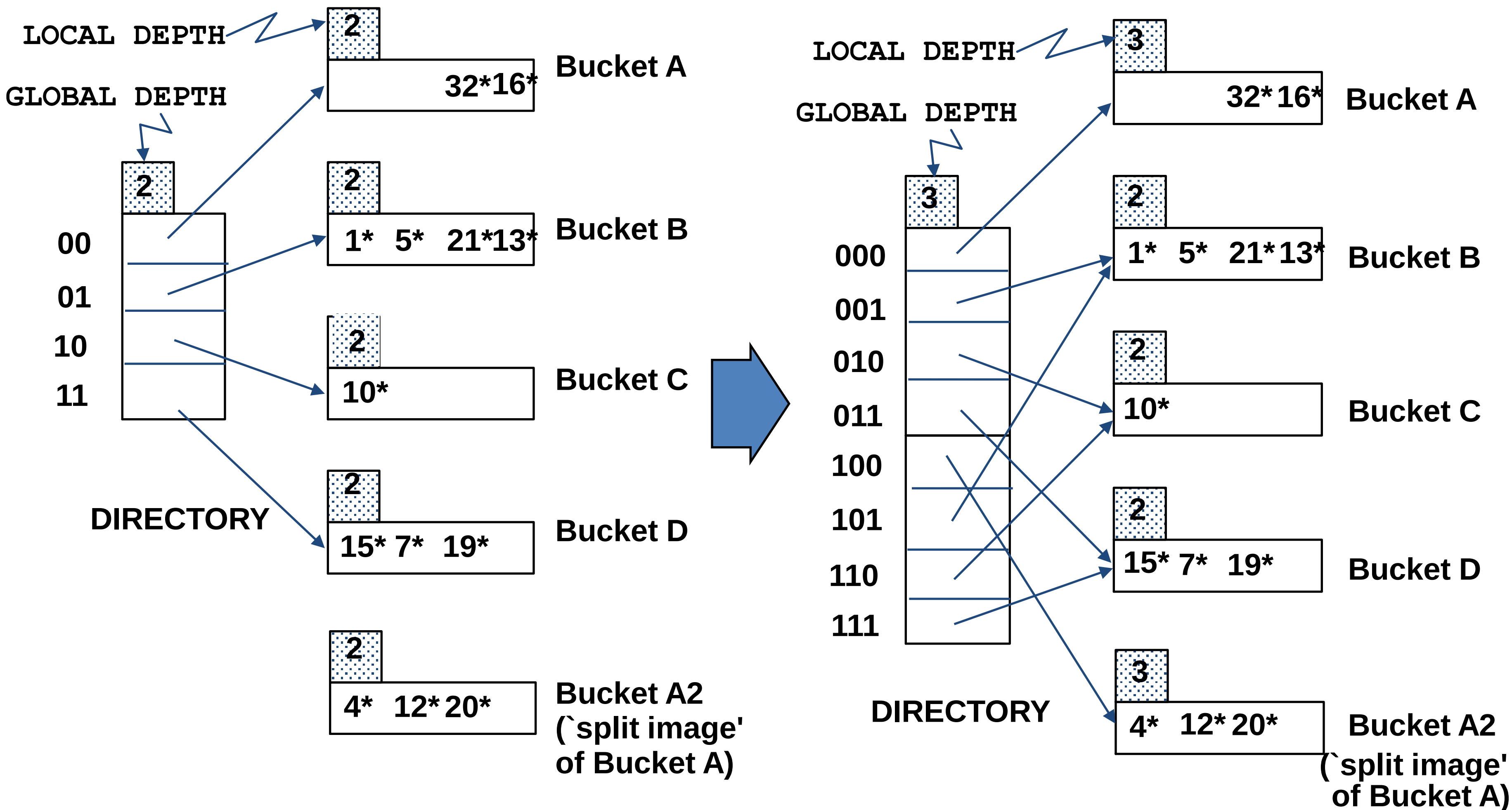


- Directory is array of size 4.
- To find bucket for *r*, take last \`*global depth*' # bits of **h**(*r*); we denote *r* by **h**(*r*).
  - If **h**(*r*) = 5 = binary 101, it is in bucket pointed to by 01.

❖ **Insert**: If bucket is full, *split* it (*allocate new page, re-distribute*).

❖ *If necessary*, double the directory. (As we will see, splitting a bucket does not always require doubling; we can tell by comparing *global depth* with *local depth* for the split bucket.)

Insert h(r)=20 (Causes Doubling)



Extendible hashing details

- 20 = binary 10100. Last **2** bits (00) tell us *r* belongs in A or A2. Last **<u>3</u>** bits needed to tell which.
  - *Global depth of directory*: Max # of bits needed to tell which bucket an entry belongs to.
  - *Local depth of a bucket*: # of bits used to determine if an entry belongs to this bucket.
- When does bucket split cause directory doubling?
  - Before insert, *local depth* of bucket = *global depth*. Insert causes *local depth* to become > *global depth*; directory is doubled by *copying it over* and `fixing' pointer to split image page. (Use of least significant bits enables efficient doubling via copying of directory!)

Linear Hashing

- LH handles the problem of long overflow chains without using a directory, and handles duplicates.
- *Idea*: Use a family of hash functions $\mathbf{h}_0$, $\mathbf{h}_1$, $\mathbf{h}_2$, ...
  - $\mathbf{h}_i(key) = \mathbf{h}(key) \bmod(2^i N)$; N = initial # buckets
  - **h** is some hash function (range is *not* 0 to N-1)
  - If $N = 2^{d0}$, for some *d0*, $\mathbf{h}_i$ consists of applying **h** and looking at the last *di* bits, where $di = d0 + i$.
  - $\mathbf{h}_{i+1}$ doubles the range of $\mathbf{h}_i$ (similar to directory doubling)

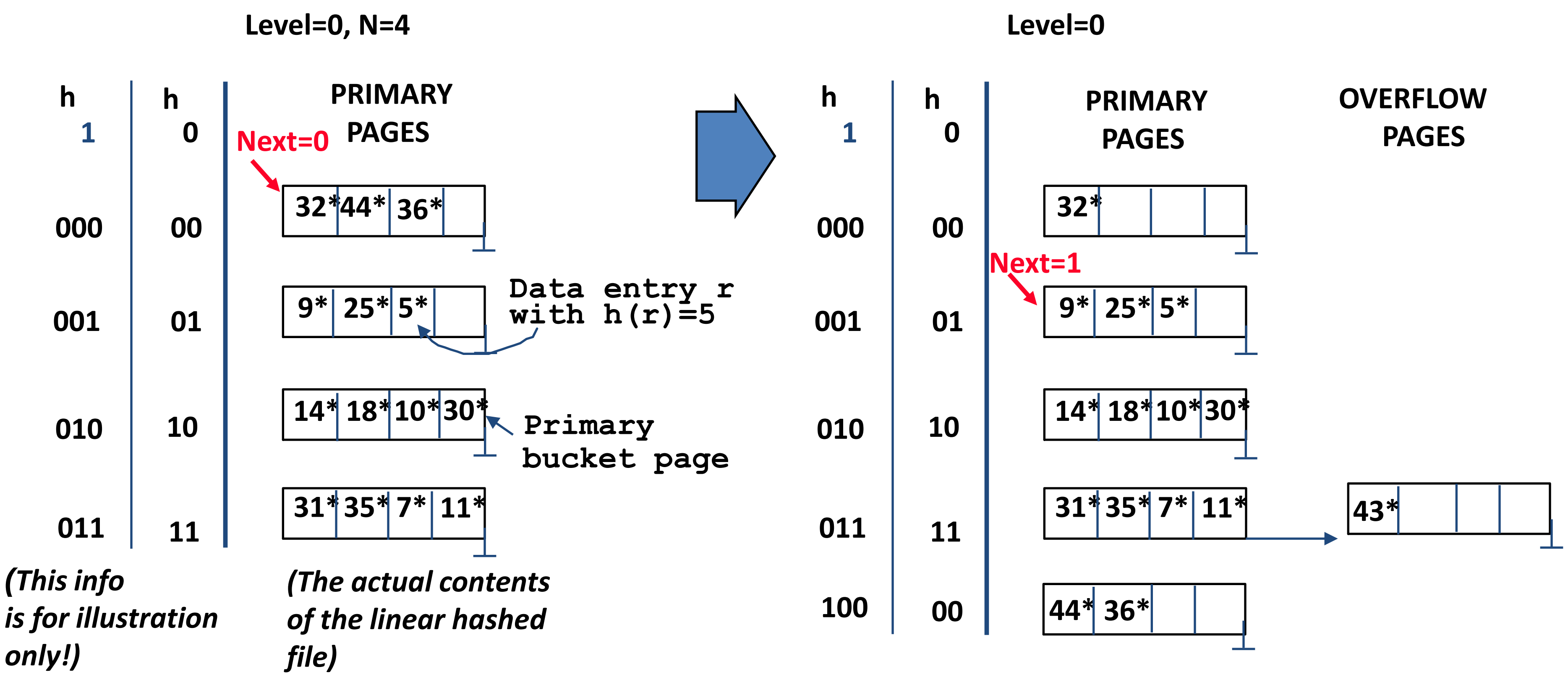
Linear Hashing (Contd.)

- Directory avoided in LH by using overflow pages, and choosing bucket to split round-robin.
  - Splitting proceeds in `rounds'. Round ends when all N_R initial (for round R) buckets are split. Buckets 0 to *Next-1* have been split; *Next* to N_R yet to be split.
  - Current round number is *Level*.
  - **Search:** To find bucket for data entry *r,* find $\mathbf{h}_{Level}(r)$:
    - If $\mathbf{h}_{Level}(r)$ in range `*Next* to N_R', *r* belongs here.
    - Else, r could belong to bucket $\mathbf{h}_{Level}(r)$ or bucket $\mathbf{h}_{Level}(r) + N_R$; must apply $\mathbf{h}_{Level+1}(r)$ to find out.

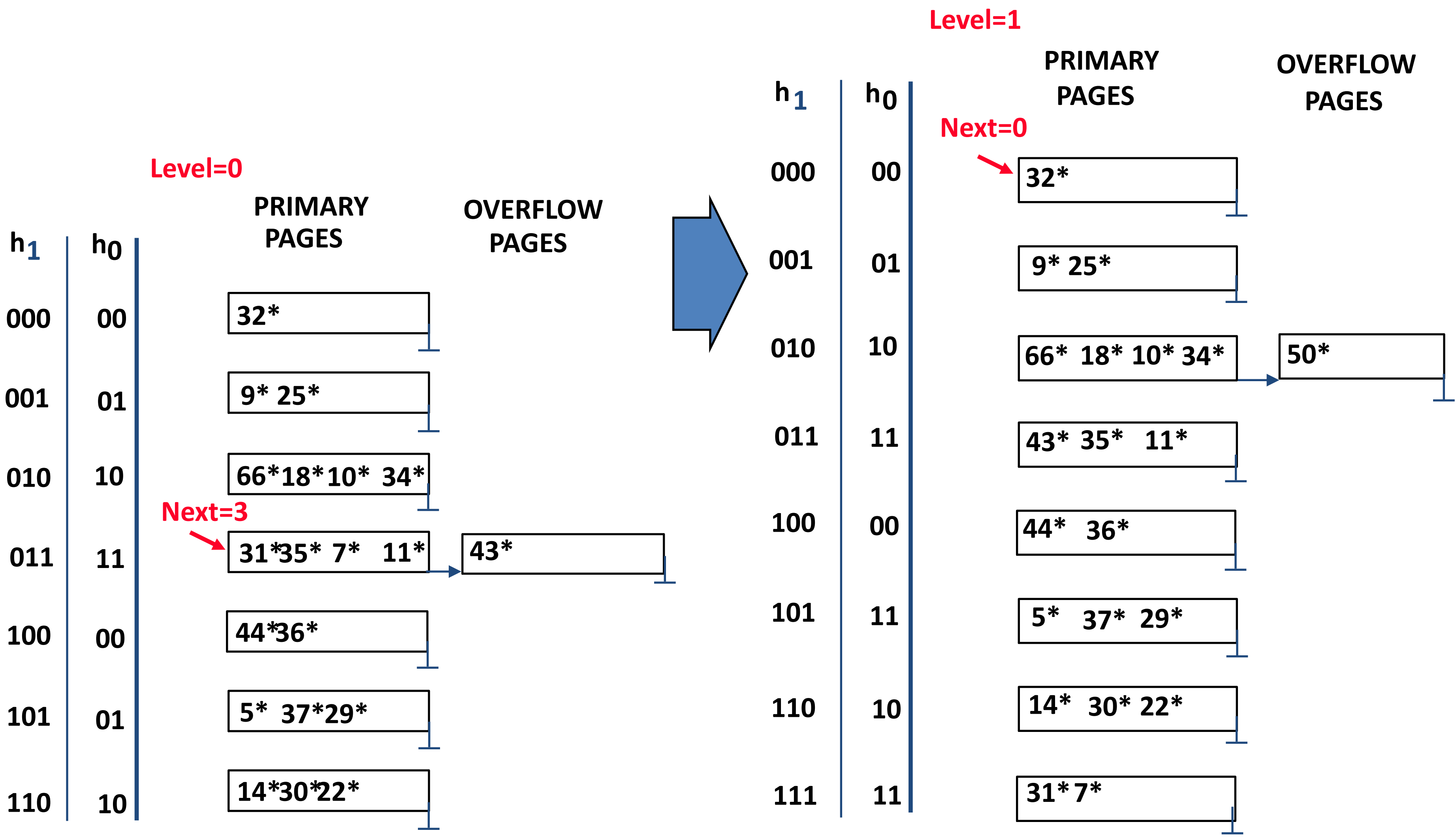
Example of Linear Hashing

- On split, **h**$_{Level+1}$ is used to redistribute entries.

Insert record with h(key) = 43*



Example: End of a Round



Summary: Hash-Based Indexes

- Hash-based indexes: best for equality searches, cannot support range searches.
- Static Hashing can lead to long overflow chains.
- Extendible Hashing avoids overflow pages by splitting a full bucket when a new data entry is to be added to it. (*Duplicates may require overflow pages.*)
  - Directory to keep track of buckets, doubles periodically.
  - Can get large with skewed data; additional I/O if this does not fit in main memory.

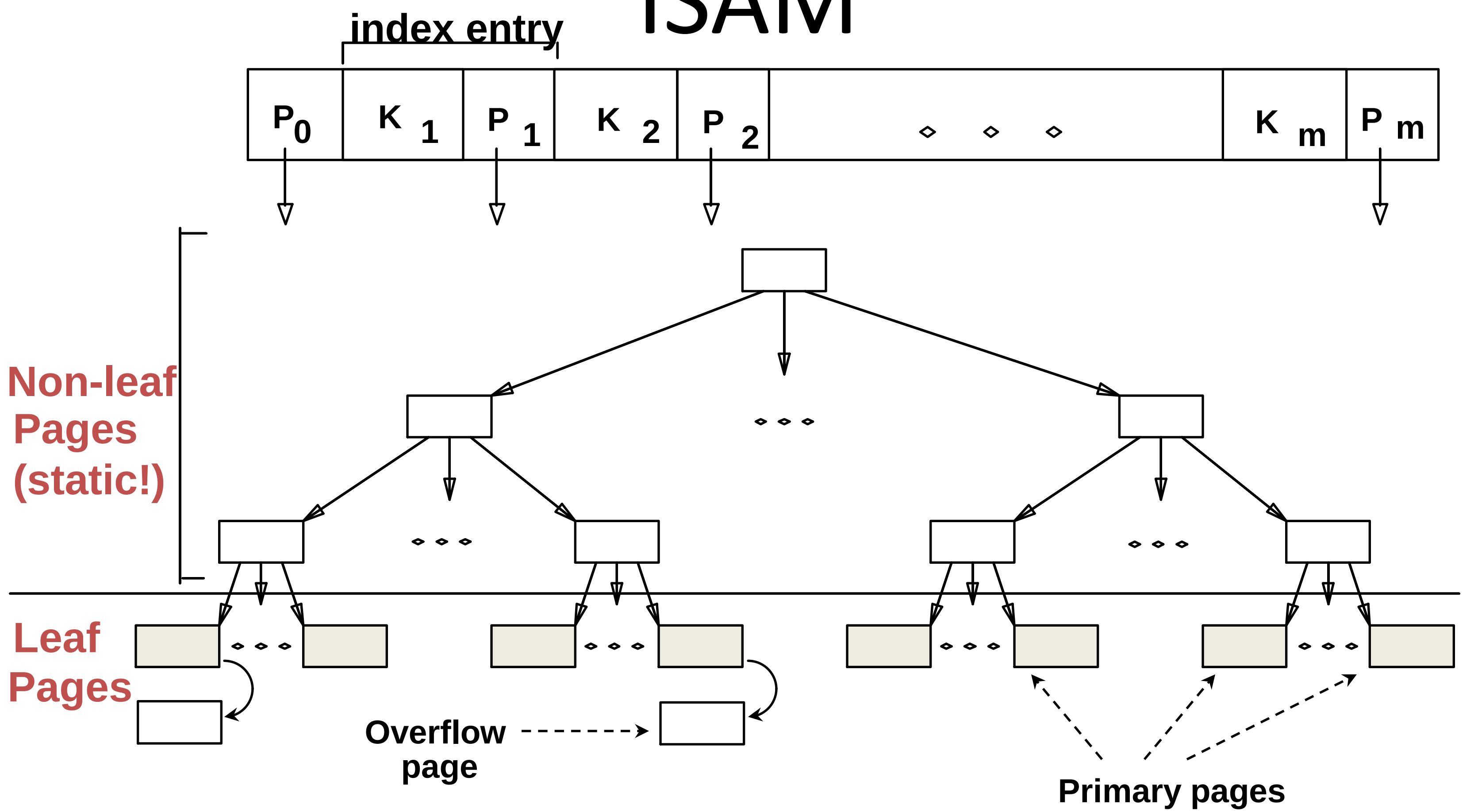
Summary: Linear hashing

- Linear Hashing avoids directory by splitting buckets round-robin, and using overflow pages.
  - Overflow pages not likely to be long.
  - Duplicates handled easily.
  - Space utilization could be lower than Extendible Hashing, since splits not concentrated on `dense' data areas.
    - Can tune criterion for triggering splits to trade-off slightly longer chains for better space utilization.
- For hash-based indexes, a *skewed* data distribution is one in which the *hash values* of data entries are not uniformly distributed!

Tree Structured Indexes

- Tree-structured indexing techniques support both *range searches* and *equality searches*.
- Tree structures with search keys on *value-based domains*
  - *ISAM*: static structure
  - *B+ tree*: dynamic, adjusts gracefully under inserts and deletes.

ISAM



- Leaf pages contain sorted data records (e.g., Alt 1 index).
- Non-leaf part directs searches to the data records; static once built!
- Inserts/deletes: use overflow pages, bad for frequent inserts.

Comments on ISAM

- Main problem
  - *Long overflow chains* after many inserts, high I/O cost for retrieval.
- Advantages
  - Simple when updates are rare.
  - Leaf pages are allocated in sequence, leading to *sequential I/O*.
  - **Non-leaf pages are static; for *concurrent access*, no need to lock non-leaf pages**
- Good performance for frequent updates?

B+tree!

B-tree Organization

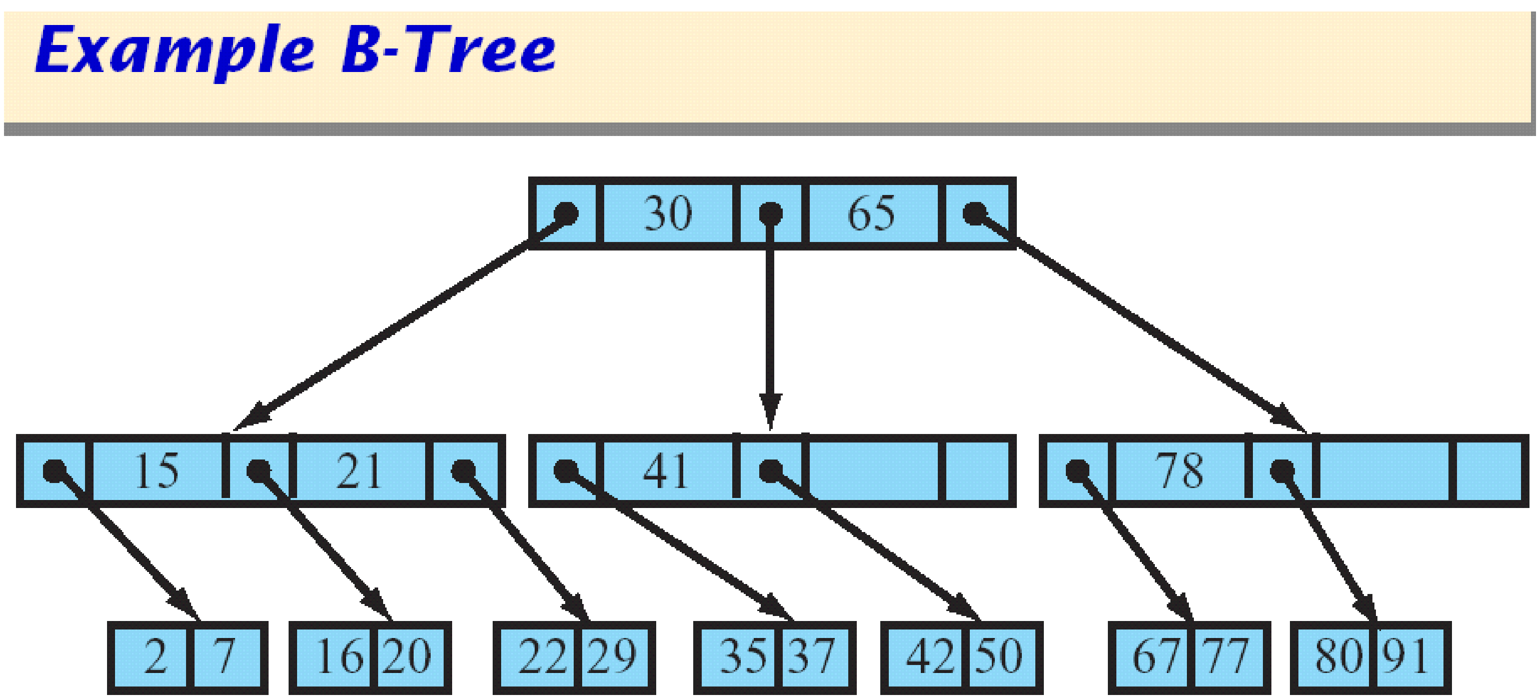
A B-tree helps minimize access to the index / directory

A B-tree is a tree where:

- Each node contains s slots for a index record and s + 1 pointers
- Each node is always at least ½ full

Order: the maximum number of keys in a non-leaf node

Fanout of a node x: the number of assigned pointers out of the node x

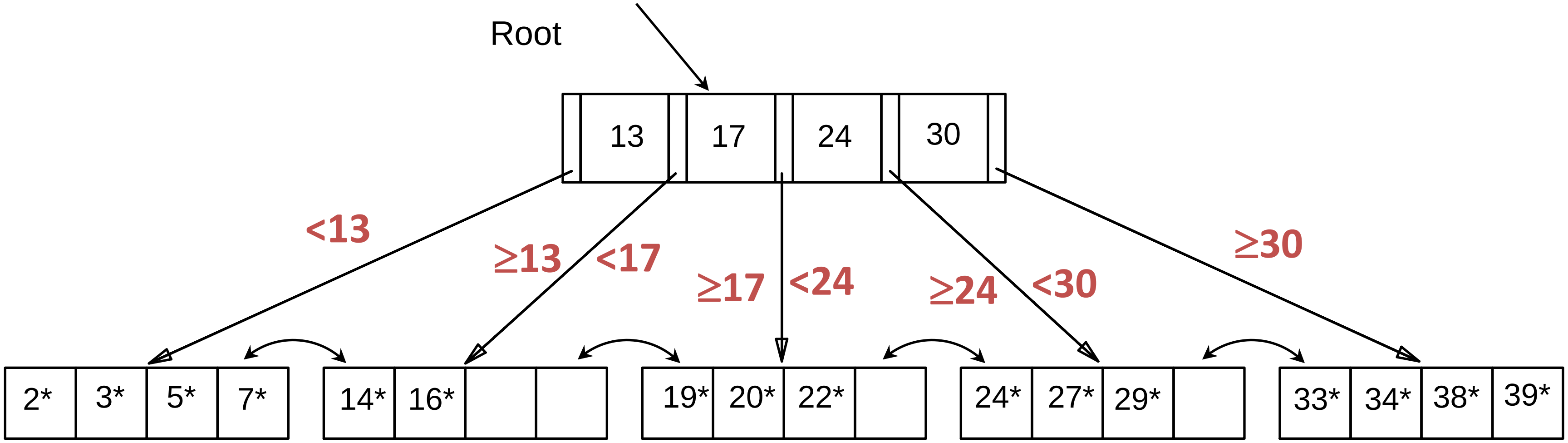


Definition of B+ Tree

- A B-tree of order *n* is a height-balanced tree , where each node may have up to *n* children, and in which:
  - All leaves (leaf nodes) are on the same level
  - No node can contain more than *n* children
  - All nodes except the root have at least n/2 children
  - The root is either a leaf node, or it has at least n/2 children

Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf (as in ISAM).
- Search for 5*, 15*, all data entries >= 24* ...



Inserting a Data Entry into a B+ Tree

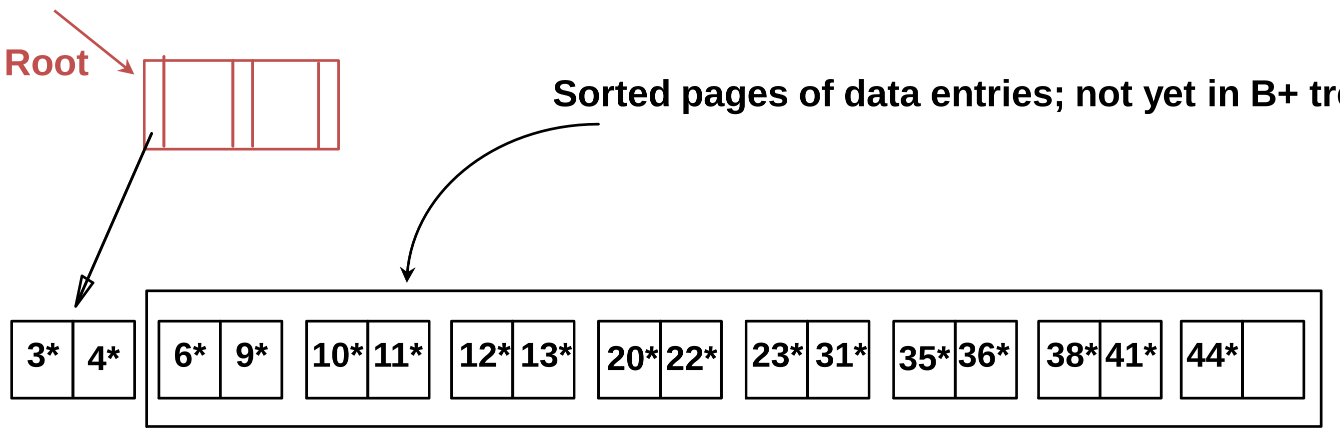
- Find correct leaf *L.*
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Redistribute entries evenly, ***copy up*** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - To 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.*

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 $\lceil \mathbf{n/2} \rceil$ **- 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.

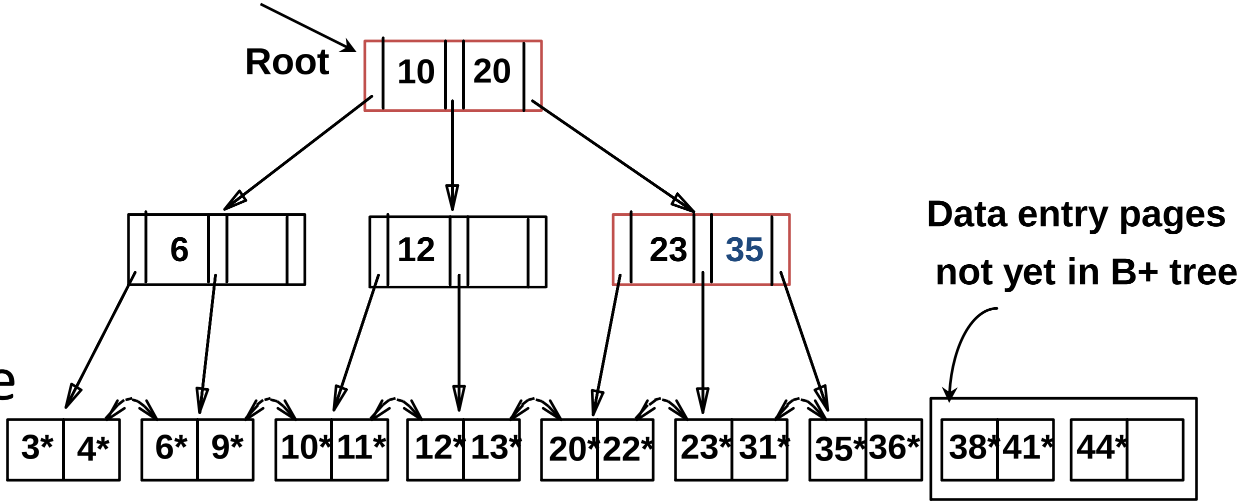
Bulk Loading Algorithm

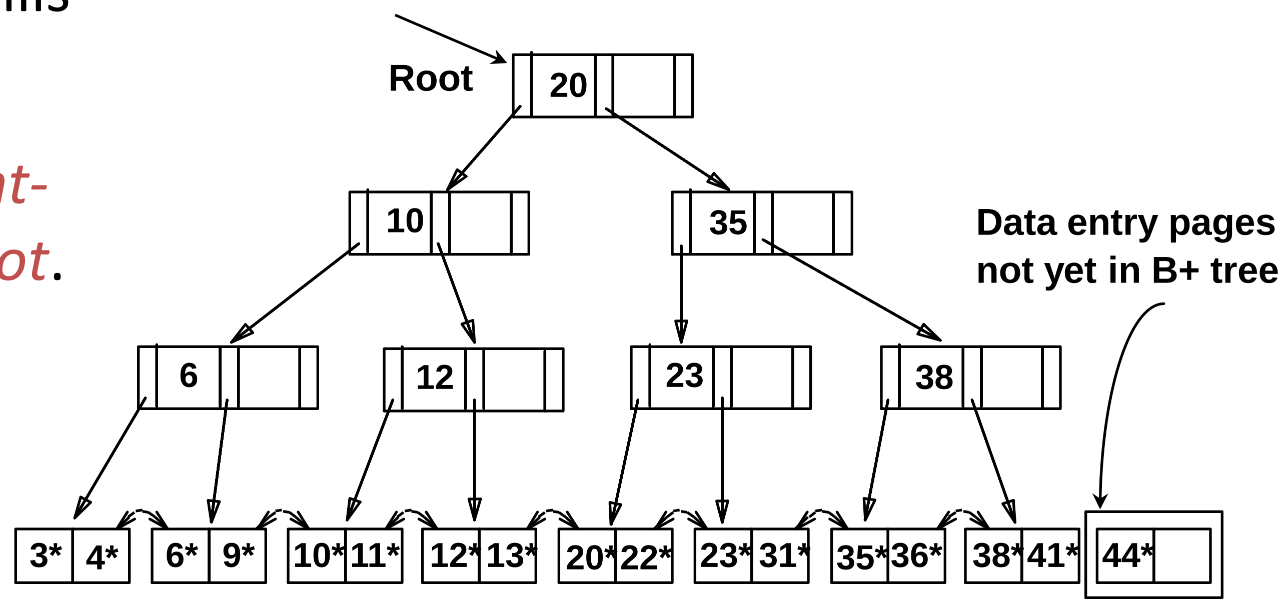
- *Initialization*:
  - Sort all data entries
  - Insert pointer to the first (leaf) page in a new (root) page.



Bulk Loading Algorithm (Contd.)

- Index entries for leaf pages always enter into r*, right-most index page just above leaf level.
- When the r* node fills up, it splits.
- Split may go up *right-most path to the root*.





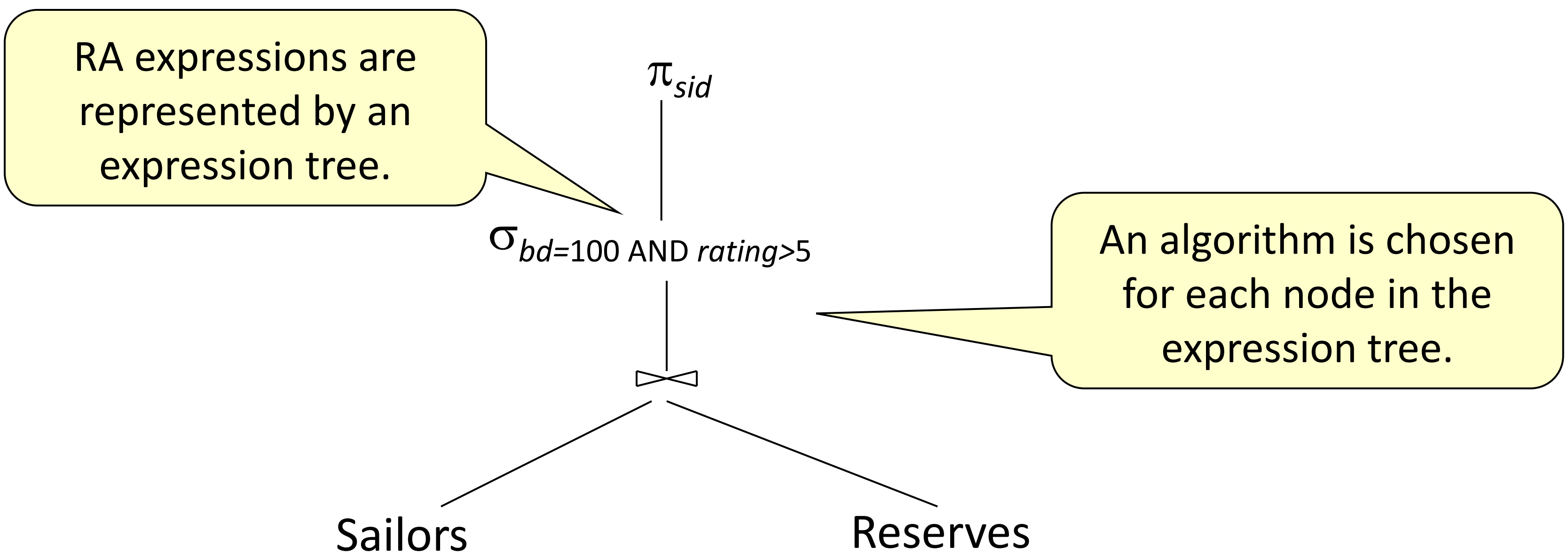
QUERY EVALUATION AND QUERY OPTIMIZATION

Tree of relational operators

Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (*sid*: integer, *bid*: integer, *day*: date, *rname*: string)

```
SELECT sid
FROM Sailors NATURAL JOIN Reserves
WHERE bid = 100 AND rating > 5;
```

$\pi_{sid}\,(\sigma_{bid=100 \text{ AND } rating>5}\ (\text{Sailors} \bowtie \text{Reserves}))$



Approaches to Evaluation

- Algorithms for evaluating relational operators use some simple ideas extensively:
  - Indexing: Can use WHERE conditions to retrieve small set of tuples (selections, joins)
  - Iteration: Sometimes, faster to scan all tuples even if there is an index. (And sometimes, we can scan the data entries in an index instead of the table itself.)
  - Partitioning: By using sorting or hashing, we can partition the input tuples and replace an expensive operation by similar operations on smaller inputs.

Relational Operations

- Operators to implement:
  - *Selection* (σ) Selects a subset of rows from relation.
  - *Projection* (π) Deletes unwanted columns from relation.
  - *Join* ($\bowtie$) Allows us to combine two relations.
  - *Set-difference* ($-$) Tuples in reln. 1, but not in reln. 2.
  - *Union* ($\cup$) Tuples in reln. 1 and in reln. 2.
  - *Aggregation* (SUM, MIN, etc.) and GROUP BY
  - *Order By* Returns tuples in specified order.
- Since each op returns a relation, ops can be *composed*. After we cover the operations, we will discuss how to *optimize* queries formed by composing them.

JOIN Algorithms

- Block Nested Loop Join
- Index Nested Loop
- Sort Merge Join

Project functionality other Algorithms

  – Influences sorting and hashing

Select functionality

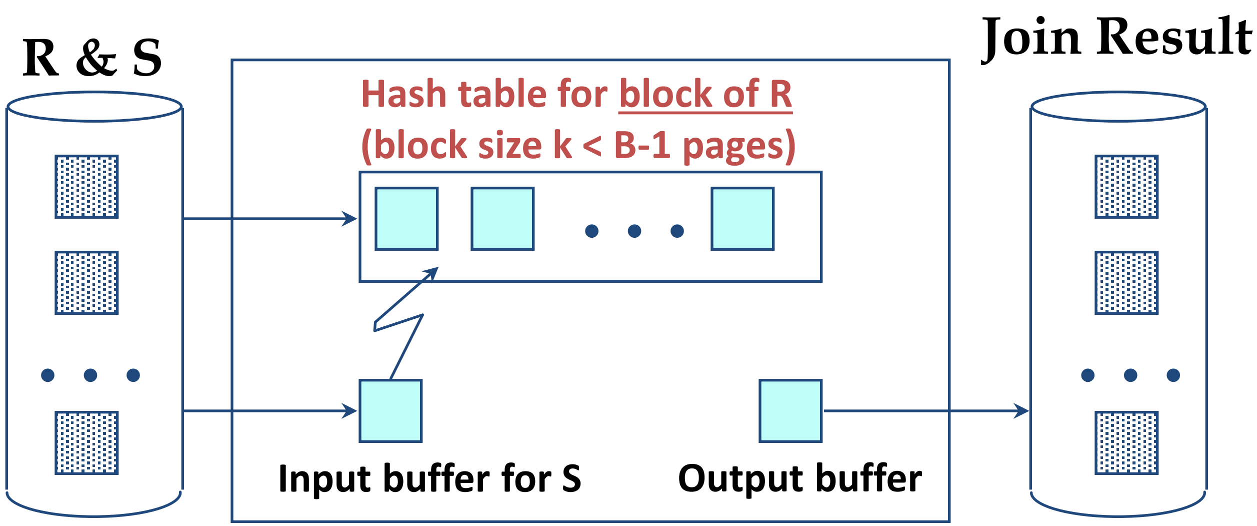
  – General selection criteria
  – Answering question via record ids

Block Nested Loops Join

- How can we utilize additional buffer pages?
  - If the smaller relation fits in memory, use it as outer, read the inner only once.
  - Otherwise, read a big chunk of it each time, resulting in reduced # times of reading the inner.
- Block Nested Loops Join:
  - Take the smaller relation, say R, as outer, the other as inner.
  - Buffer allocation: one buffer for scanning the inner S, one buffer for output, all remaining buffers for holding a ``block'' of outer R.

Block Nested Loops Join Diagram

```
foreach block  in R do
   build a hash table on R-block
   foreach S page
         for each matching tuple r in R-block, s in S-page do
            add <r, s> to result
```



Examples of Block Nested Loops

- Cost: Scan of outer table + #outer blocks * scan of inner table
  - #outer blocks = $\lceil$ # pages of outer / block size $\rceil$
  - Given available buffer size B, block size is at most B-2.
- With Sailors (S) as outer, a block has 100 pages of S:
  - Cost of scanning S is 500 I/Os; a total of 5 *blocks*.
  - Per block of S, we scan Reserves; 5*1000 I/Os.
  - Total = 500 + 5 * 1000 = 5,500 I/Os.

- Sailors:
  - Each tuple is 50 bytes long,
  - 80 tuples per page,
  - 500 pages.
- Reserves:
  - Each tuple is 40 bytes long,
  - 100 tuples per page,
  - 1000 pages.

Index Nested Loops Join

```
foreach tuple r in R do
        foreach tuple s in S where r_i == s_j do
                add <r, s> to result
```

- If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.
  - Cost: $M + ((M*p_R)$ * cost of finding matching S tuples)
- For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding S tuples (assuming Alt. (2) or (3) for data entries) depends on clustering.
  - Clustered index: 1 I/O (typical).
  - Unclustered: up to 1 I/O per matching S tuple.

Sort-Merge Join $(R \bowtie_{i=j} S)$

- Sort R and S on join column using external sorting.
- Merge R and S on join column, output result tuples.

  Repeat until either R or S is finished:
  - *Scanning*:
    - Advance scan of R until current R-tuple >=current S tuple,
    - Advance scan of S until current S-tuple>=current R tuple;
    - Do this until current R tuple = current S tuple.
  - *Matching*:
    - Match all R tuples and S tuples with same value; output <r, s> for all pairs of such tuples.
- Data access patterns for R and S?

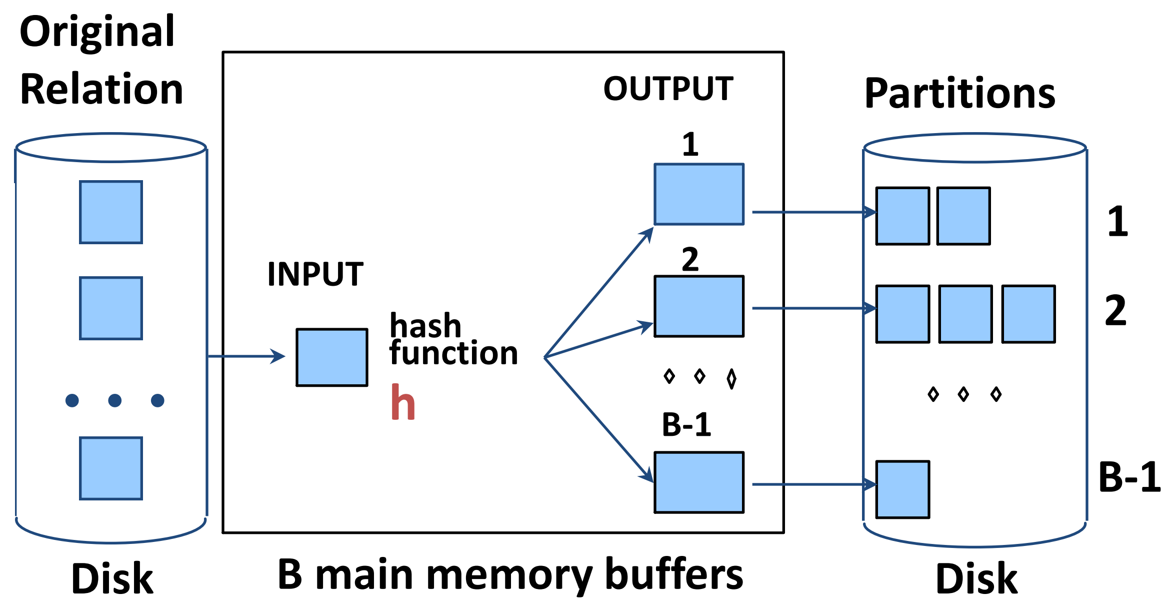
R is scanned once, each S partition scanned once per matching R tuple

Refinement of Sort-Merge Join

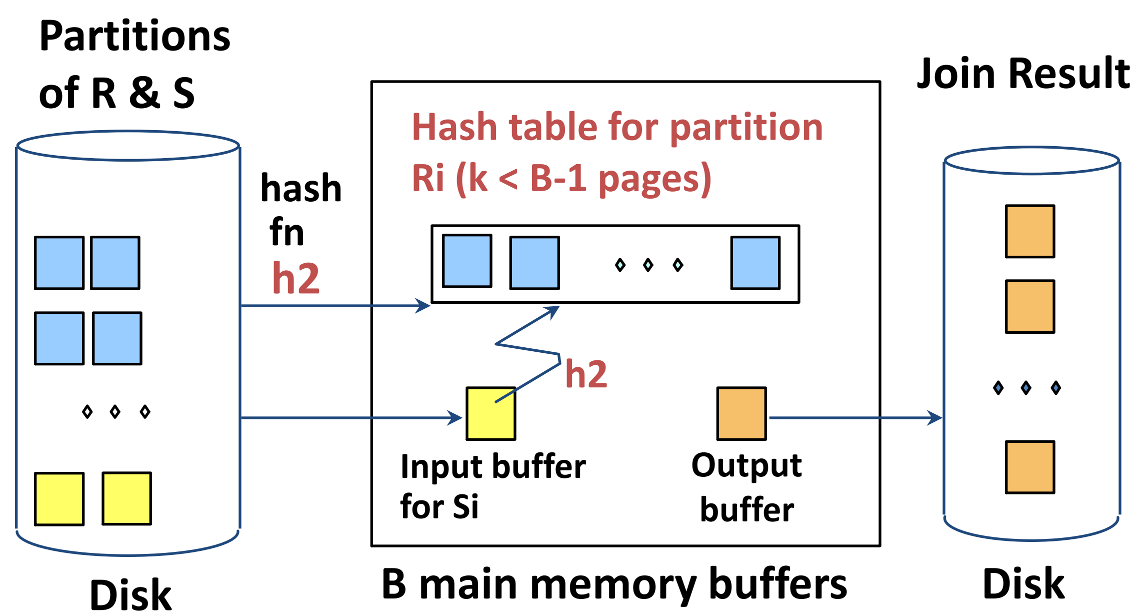
- *Idea*:
  - *Sorting* of R and S has respective merging phases
  - *Join* of R and S also has a merging phase
  - Combine all these merging phases!
- Two-pass algorithm for sort-merge join:
  - Pass 0: sort subfiles of R, S individually
  - Pass 1: merge sorted runs of R, merge sorted runs of S, and merge the resulting R and S files as they are generated by checking the join condition.

Hash Join

- Partitioning: Partition both relations using hash fn **h**: Ri tuples will only match with Si tuples.



- Probing: Read in partition i of R, build hash table on Ri using **h2 (<> h!)**. Scan partition i of S, search for matches.



Approach 1 to General **Selections**

- (1) Find the *most selective access path*, *retrieve* tuples using it, and (2) apply any remaining terms that don't match the index *on the fly*.
  - *Most selective access path:* An index or file scan that is expected to require the smallest # I/Os.
    - Terms that match this index reduce the number of tuples *retrieved*;
    - Other terms are used to discard some retrieved tuples, but do not affect I/O cost.
  - Consider *day<8/9/94 AND bid=5 AND sid=3*.
    - A B+ tree index on *day* can be used; then, *bid=5* and *sid=3* must be checked for each retrieved tuple.
    - A hash index on *<bid, sid>* could be used; *day<8/9/94* must then be checked on the fly.

Approach 2: **SELECT** Intersection of Rids

- If we have 2 or more matching indexes that use Alternatives (2) or (3) for data entries:
  - Get sets of rids of data records using each matching index.
  - *Intersect* these sets of rids.
  - Retrieve the records and apply any remaining terms.
  - Consider *day<8/9/94 AND bid=5 AND sid=3*. If we have a B+ tree index on *day* and an index on *sid*, both using Alternative (2), we can:
    - retrieve rids of records satisfying *day<8/9/94* using the first, rids of records satisfying *sid=3* using the second,
    - intersect these rids,
    - retrieve records and check *bid=5.*

Projection Based on Sorting

- Modify Pass 0 of external sort to eliminate unwanted fields.
  - Runs of about 2B pages are produced,
  - But tuples in runs are smaller than input tuples. (Size ratio depends on # and size of fields that are dropped.)
- Modify merging passes to eliminate duplicates.
  - # result tuples smaller than input. Difference depends on # of duplicates.
- Cost: In Pass 0, read input relation (size M), write out same number of smaller tuples. In merging passes, fewer tuples written out in each pass.
  - Using Reserves example, 1000 input pages reduced to 250 in Pass 0 if size ratio is 0.25.

Projection Based on Hashing

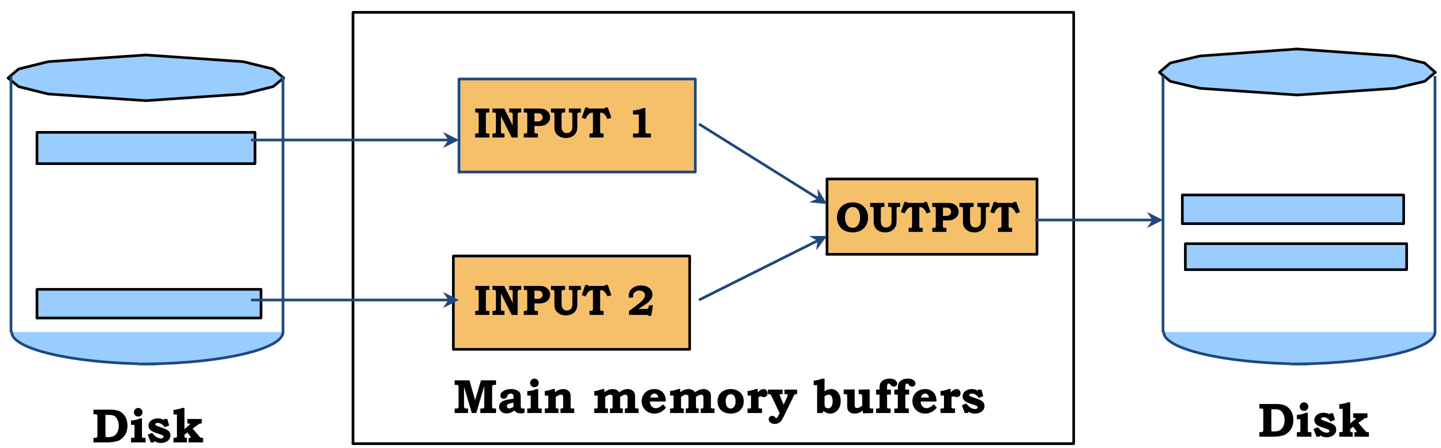
- *Partitioning phase*: Read R using one input buffer. For each tuple, discard unwanted fields, apply hash function *h1* to choose one of B-1 output buffers.
  - Result is B-1 partitions (of tuples with no unwanted fields). 2 tuples from different partitions guaranteed to be distinct.
- *Duplicate elimination phase*: For each partition, read it and build an in-memory hash table, using hash fn *h2* (<> *h1*) on all fields, while discarding duplicates.
  - If partition does not fit in memory, can apply hash-based projection algorithm recursively to this partition.
- Cost: For partitioning, read R, write out each tuple, but with fewer fields. This is read in next phase.

EXTERNAL SORT

2-Way Sort: Requires 3 Buffers

- Pass 1: Read a page, sort it, write it.
  - only one buffer page is used
- Pass 2, 3, …, etc.:
  - three buffer pages used.

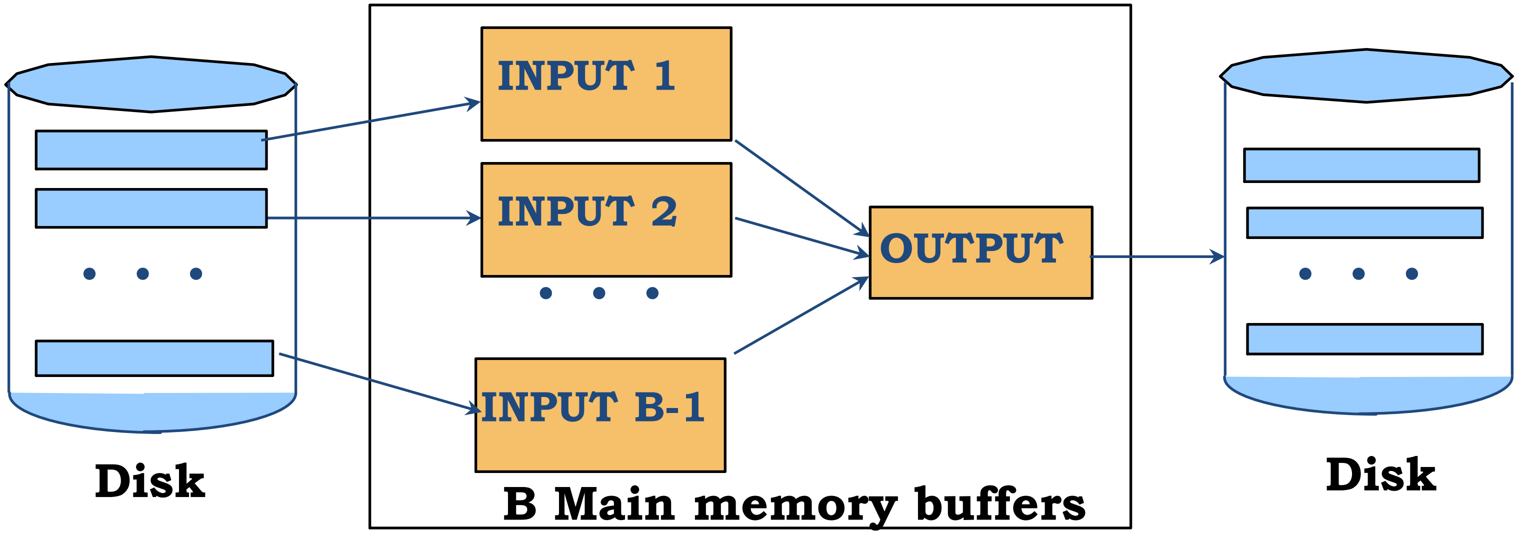
Partition data
Pass determines
Size of partition



General External Merge Sort

More than 3 buffer pages. How can we utilize them?

- 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 2, 3…, etc.: merge *B-1* runs.



Cost of External Merge Sort

- E.g., with 5 (B) buffer pages, sort 108 (N) page file:

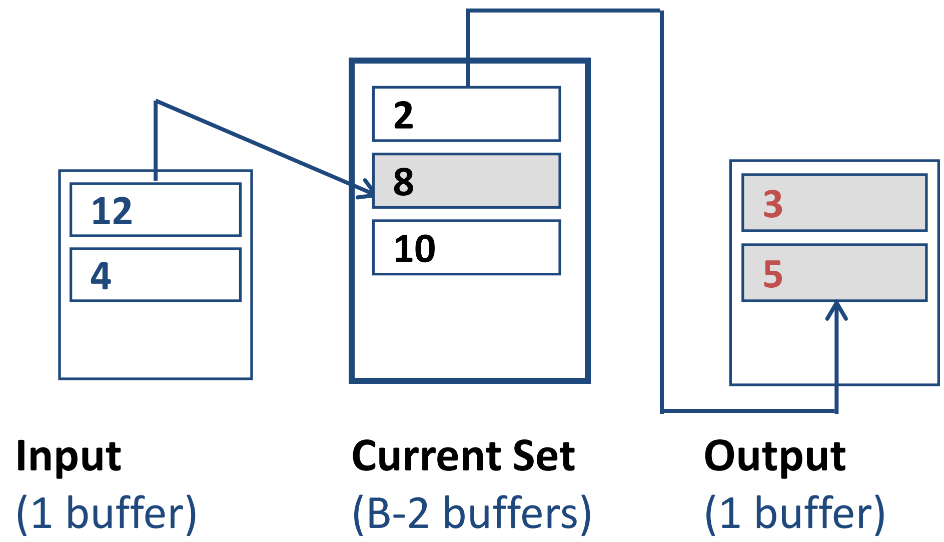
Pass 0	$\lceil 108/5 \rceil$ = 22 sorted runs of 5 pages each (last run is only 3 pages)	$\lceil N/B \rceil$ sorted runs of B pages each
Pass 1	$\lceil 22/4 \rceil$ = 6 sorted runs of 20 pages each (last run is only 8 pages)	$\lceil N/B \rceil / (B-1)$ sorted runs of B(B-1) pages each
Pass 2	2 sorted runs, 80 pages and 28 pages	$\lceil N/B \rceil / (B-1)^2$ sorted runs of $B(B-1)^2$ pages
Pass 3	Sorted file of 108 pages	$\lceil N/B \rceil / (B-1)^3$ sorted runs of $B(B-1)^3$ ($\geq N$) pages

- Number of passes = $1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$

  Cost = 2N * (# of passes)

Replacement Sort

- Organize B available buffers:
  - 1 buffer for *input*
  - B-2 buffers for *current set*
  - 1 buffer for *output*

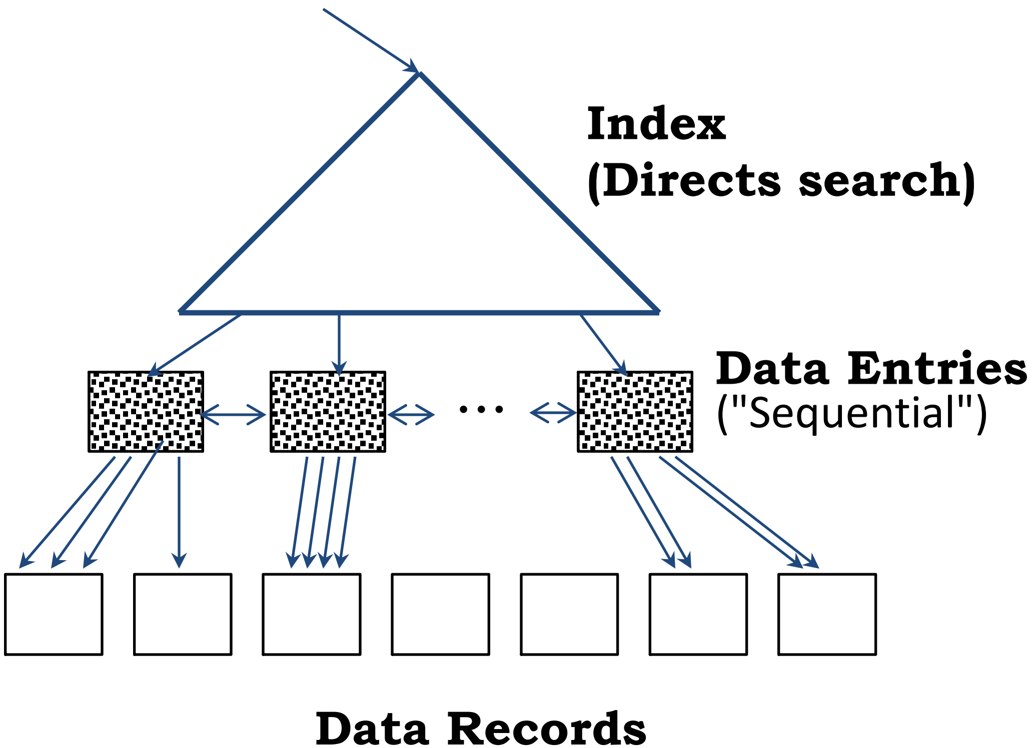


- Pick tuple *r* in the current set with the *smallest value that is ≥ largest value in output*, e.g. 8, to extend the current run.
- Fill the space in current set by adding tuples from input.
- Write output buffer out if full, extending the current run.
- Current run terminates if *every tuple in the current set is smaller than the largest tuple in output*.

Clustered B+ Tree Used for Sorting

- Cost: root to the left-most leaf, then retrieve all leaf pages (Alternative 1)

❖ If Alternative 2 is used? Additional cost of retrieving data records: each page fetched just once.

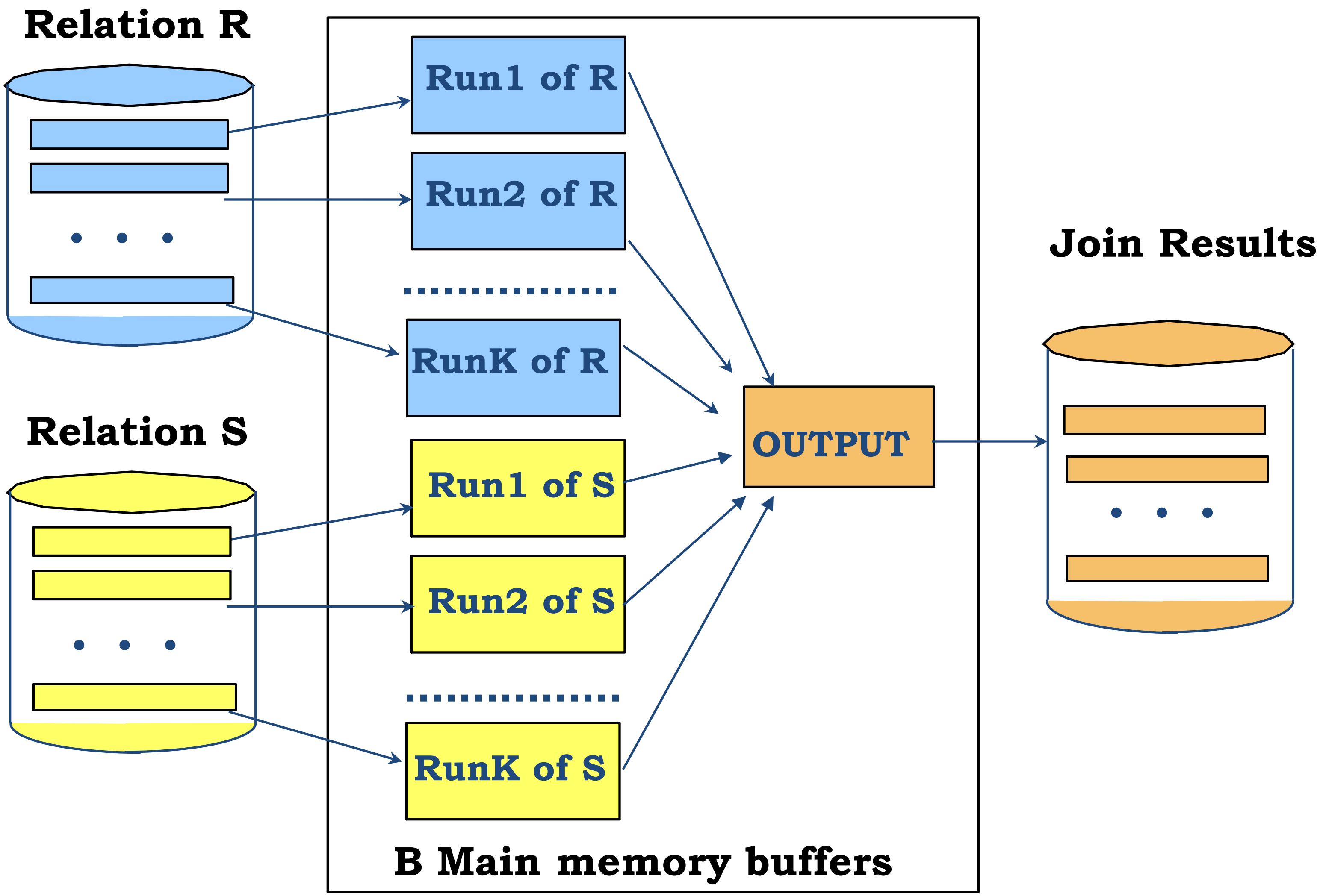


Almost always better than external sorting!

Refinement of Sort-Merge Join

- *Idea*:
  - *Sorting* of R and S has respective merging phases
  - *Join* of R and S also has a merging phase
  - Combine all these merging phases!
- Two-pass algorithm for sort-merge join:
  - Pass 0: sort subfiles of R, S individually
  - Pass 1: merge sorted runs of R, merge sorted runs of S, and merge the resulting R and S files as they are generated by checking the join condition.

2-Pass Sort-Merge Algorithm



Using an Index for Selections

- Cost depends on # qualifying tuples, and *clustering*.
  - Cost of finding data entries (often small) + cost of retrieving records (could be large w/o clustering).
  - For *gpa* > 3.0, if 10% of tuples qualify (100 pages, 10,000 tuples), cost ≈ 100 I/Os with a clustered index; otherwise, up to 10,000 I/Os!
- Important refinement for unclustered indexes:
  1. Find qualifying data entries.
  2. **Sort the rid's** of the data records to be retrieved.
  3. Fetch rids in order.

Each data page is looked at just once, although # of such pages likely to be higher than with clustering.

Approach 1 to General Selections

- (1) Find the *most selective access path*, *retrieve* tuples using it, and (2) apply any remaining terms that don't match the index *on the fly*.
  - *Most selective access path:* An index or file scan that is expected to require the smallest # I/Os.
    - Terms that match this index reduce the number of tuples *retrieved*;
    - Other terms are used to discard some retrieved tuples, but do not affect I/O cost.
  - Consider *day<8/9/94 AND bid=5 AND sid=3*.
    - A B+ tree index on *day* can be used; then, *bid=5* and *sid=3* must be checked for each retrieved tuple.
    - A hash index on *<bid, sid>* could be used; *day<8/9/94* must then be checked on the fly.

Approach 2: Intersection of Rids

- If we have 2 or more matching indexes that use Alternatives (2) or (3) for data entries:
  - Get sets of rids of data records using each matching index.
  - *Intersect* these sets of rids.
  - Retrieve the records and apply any remaining terms.
  - Consider *day<8/9/94 AND bid=5 AND sid=3*. If we have a B+ tree index on *day* and an index on *sid*, both using Alternative (2), we can:
    - retrieve rids of records satisfying *day<8/9/94* using the first, rids of records satisfying *sid=3* using the second,
    - intersect these rids,
    - retrieve records and check *bid=5.*

Summary: Query plan

- Many implementation techniques for each operator; no universally superior technique for most operators.
- Must consider available alternatives for each operation in a query and choose best one based on:
  - system state (e.g., memory) and
  - statistics (table size, # tuples matching value k).
- This is part of the broader task of optimizing a query composed of several ops.

Representation of a SQL Command

```
SELECT      {DISTINCT} <list of columns>
FROM        <list of relations>
{WHERE      <list of "Boolean Factors">}
{GROUP BY <list of columns>
{HAVING     <list of Boolean Factors>}}
{ORDER BY <list of columns>};
```

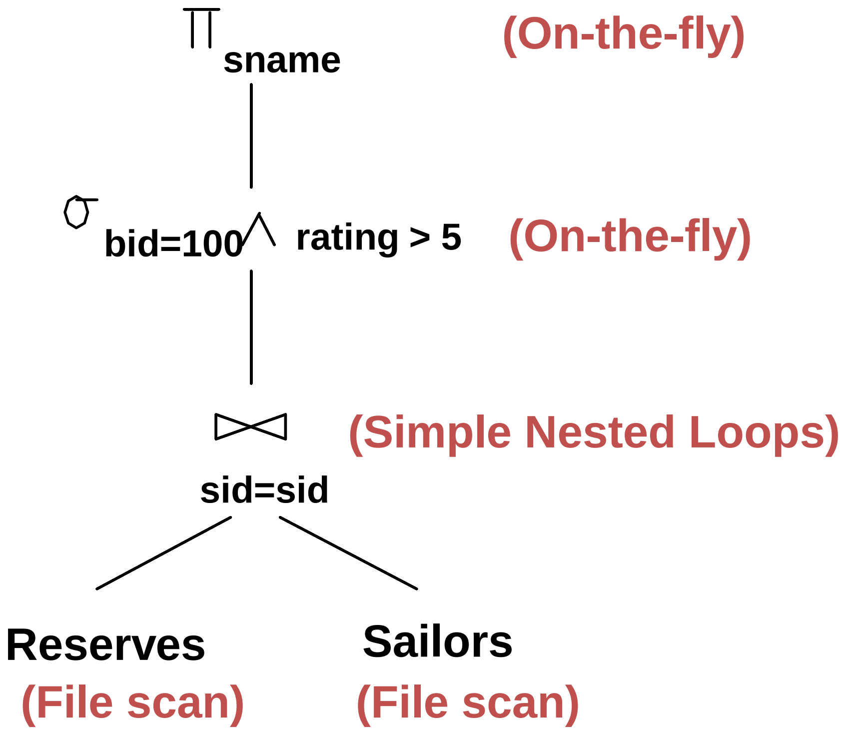
- Query Semantics:
  1. Take Cartesian product (a.k.a. cross-product) of relns in FROM, projecting only to those columns that appear in other clauses
  2. If a WHERE clause exists, apply all filters in it
  3. If a GROUP BY clause exists, form groups on the result
  4. If a HAVING clause exists, filter groups with it
  5. If an ORDER BY clause exists, make sure output is in the right order
  6. If there is a DISTINCT modifier, remove duplicates

System Catalog

- System information: buffer pool size and page size.
- For each relation:
  - relation name, file name, file structure (e.g., heap file)
  - attribute name and type of each attribute
  - index name of each index on the relation
  - integrity constraints…
- For each index:
  - index name and structure (B+ tree)
  - search key attribute(s)
- For each view:
  - view name and definition
- Statistics about each relation (R) and index (I):

Query Evaluation Plan

- *Query evaluation plan* is an extended RA tree, with additional annotations:
  - *access method* for each relation;
  - *implementation method* for each relational operator.
- Cost Approximation
- Manipulating plans:
  - *Relational Alebra Equivalence*
  - *Push selections below the join*.
  - *Materialization*: store a temporary relation T,
  - if the subsequent join needs to *scan T multiple times*.
    - The opposite is *pipelining*



Query Blocks: Units of Optimization

- An SQL query is parsed into a collection of *query blocks*, and these are optimized one block at a time.

```
SELECT  S.sname
FROM    Sailors S
WHERE  S.age IN
    (SELECT  MAX (S2.age)
      FROM  Sailors S2
      GROUP BY  S2.rating)
```

Outer block *Nested block*

- Nested blocks are usually treated as calls to a subroutine, made once per outer tuple.

Cost Estimation for Multi-relation Plans

```
SELECT attribute list
FROM relation list
WHERE term1 AND ... AND termk
```

- Consider a query block:
- *Reduction factor (RF)* is associated with each *term*.
- *Max number tuples in result* = the product of the cardinalities of relations in the FROM clause.
- *Result cardinality* = max # tuples * product of all RF's.
- Multi-relation plans are built up by joining one new relation at a time.
  - Cost of join method, plus estimate of join cardinality gives us both cost estimate and result size estimate.

Query Optimization: Summary

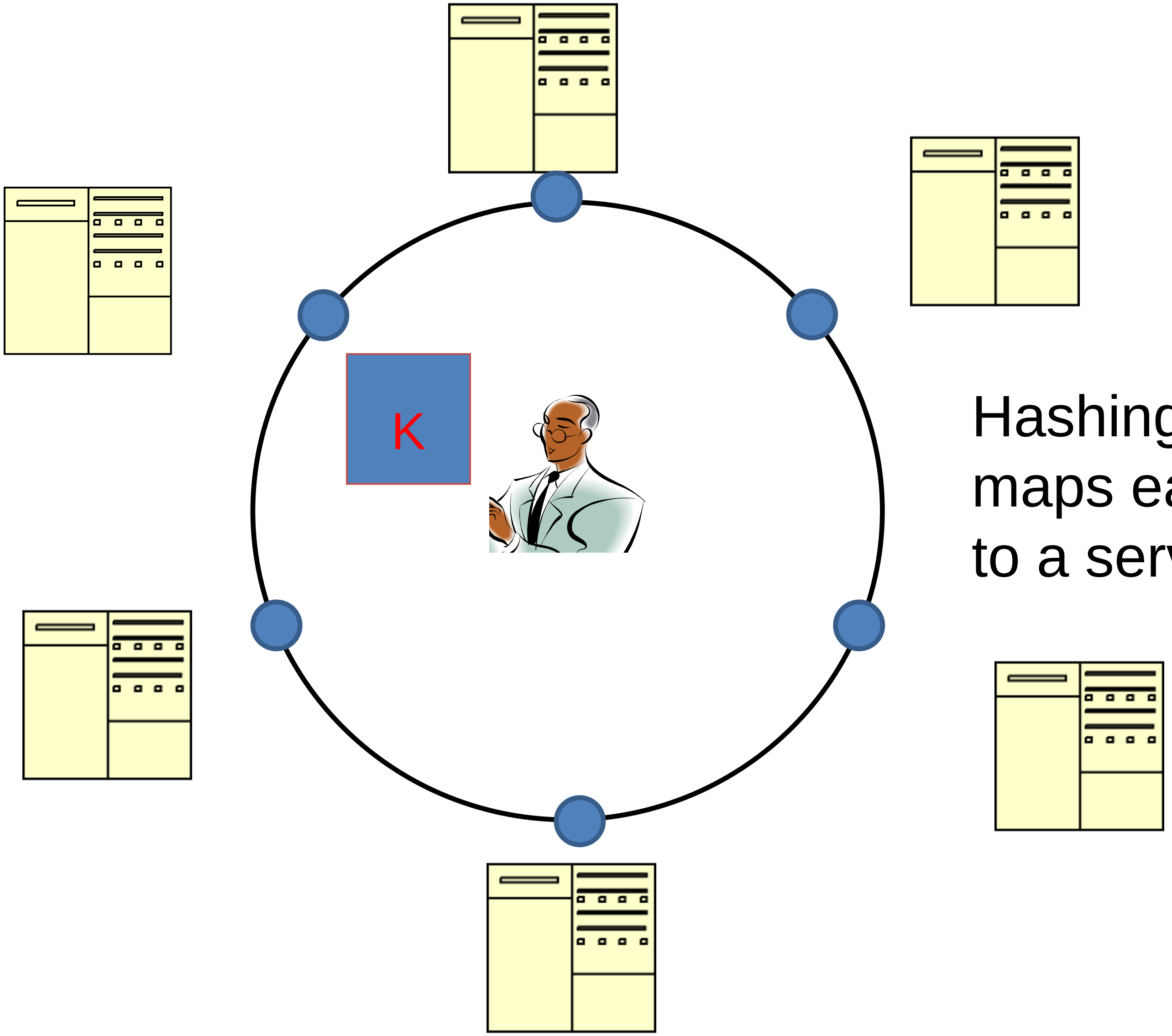
- Two parts to optimizing a query:
  - Consider a set of alternative plans.
    - Must prune search space; typically, left-deep plans only.
  - Must estimate cost of each plan that is considered.
    - Must estimate size of result and cost for each plan node.
    - *Key issues*: Statistics, indexes, operator implementations.

Query Optimization: Summary

- Single-relation queries:
  - All access paths considered, cheapest is chosen.
  - *Issues*: Selections that *match* index, whether index key has all needed fields and/or provides tuples in a desired order.
- Multiple-relation queries:
  - All single-relation plans are first enumerated.
    - Selections/projections considered as early as possible.
  - Next, for each 1-relation plan, all ways of joining another relation (as inner) are considered.
  - Next, for each 2-relation plan that is `retained', all ways of joining another relation (as inner) are considered, etc.
  - At each level, for each subset of relations, only best plan for each interesting order of tuples is `retained'.

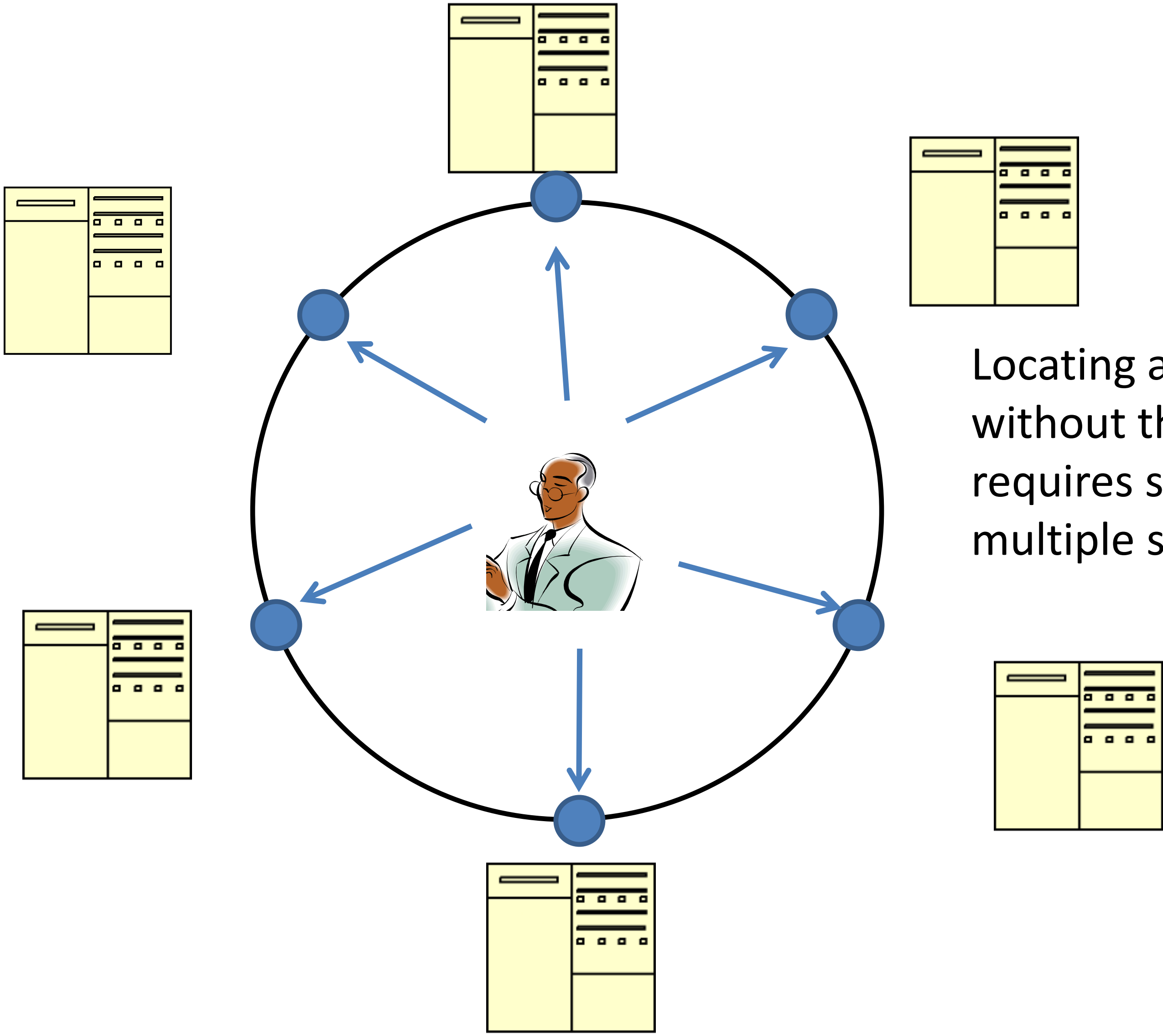
NO SQL

Typical NoSQL architecture



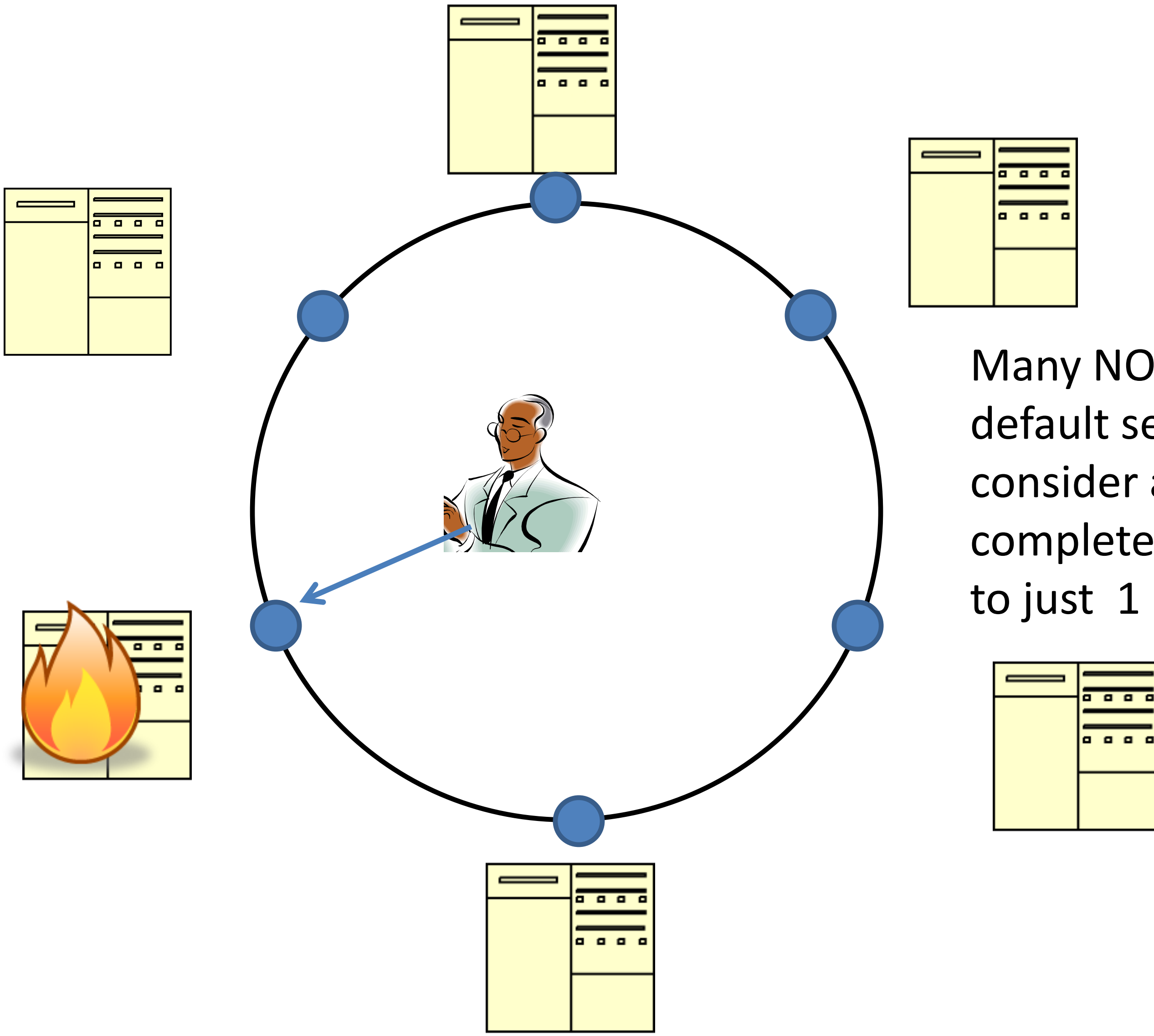
Hashing function maps each key to a server

The search problem: No Hash key



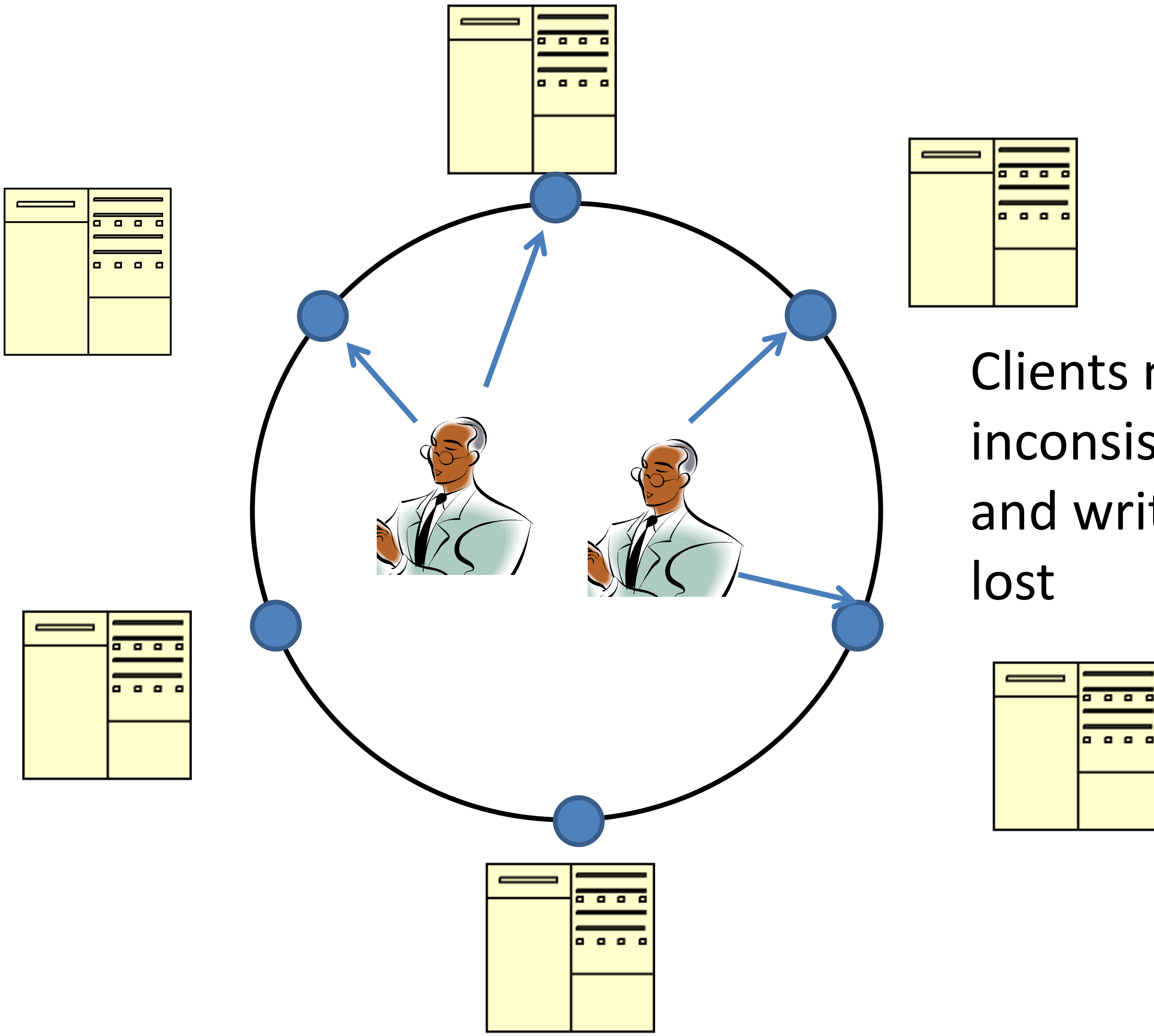
Locating a record without the hash key requires searching multiple servers

The Fault Tolerance problem



Many NOSQL system's default settings consider a write complete after writing to just 1 node

The consistency problem



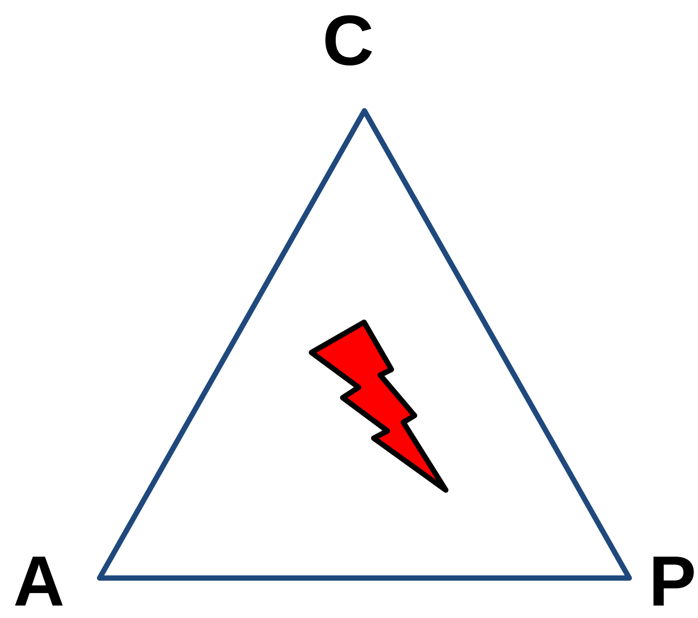
Clients may read inconsistent data and writes may be lost

Theory of NOSQL: CAP

GIVEN:

- Many nodes
- Nodes contain ***replicas of partitions*** of the data

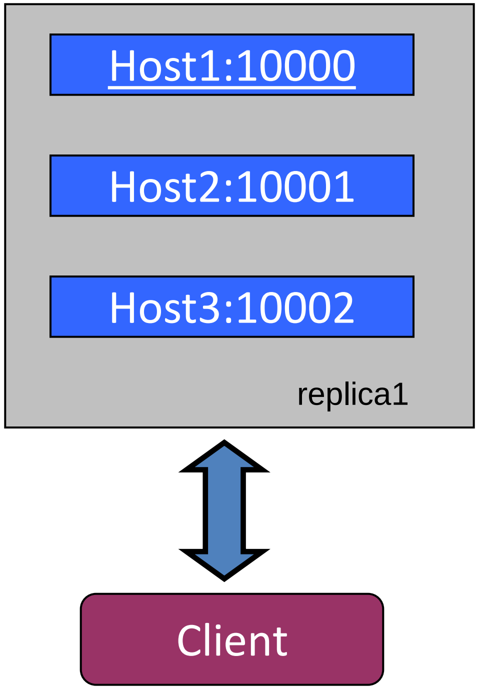
- **Consistency**
  - all replicas contain the same version of data
- **Availability**
  - system remains operational on failing nodes
- **Partition tolarence**
  - multiple entry points
  - system remains operational on system split



CAP Theorem: satisfying all three at the same time is impossible

Replica Sets

- Redundancy and Failover
- Zero downtime for upgrades and mainentance



- Master-slave replication
  - Strong Consistency
  - Delayed Consistency

- Geospatial features

How does it vary from SQL?

- Looser schema definition
- Various schema models
  - Key value pair
  - Document oriented
  - Graph
  - Column based
- Applications written to deal with specific documents
  - Applications aware of the schema definition as opposed to the data
- Designed to handle distributed, large databases
- Trade off: ad hoc queries for speed and growth of database

ACID - BASE

Atomicity

Consistency

Isolation

Durability

⟺

Basically

Available (CP)

Soft-state

Eventually consistent (Asynchronous propagation)

Pritchett, D.: BASE: An Acid Alternative (queue.acm.org/detail.cfm?id=1394128)

What is MapReduce?

- Programming model for expressing distributed computations on massive amounts of data

AND

- An execution framework for large-scale data processing on clusters of commodity servers

Programming Model

- Transforms set of input key-value pairs to set of output key-value pairs
  - Map function written by user
  - Map: (k1, v1) → list (k2, v2)
  - MapReduce library groups all intermediate pairs with same key together
- Reduce written by user
  - Reduce: (k2, list (v2)) → list (v2)
  - Usually zero or one output value per group
  - Intermediate values supplied via iterator (to handle lists that do not fit in memory)

Execution Framework

- Handles scheduling of the tasks
  - Assigns workers to maps and reduce tasks
  - Handles data distribution
    - Moves the process to the data
  - Handles synchronization
    - Gathers, sorts and shuffles intermediate data
  - Handles faults
    - Detects worker failures and restarts
  - Understands the distributed file system

MongoDB Basics

- A MongoDB instance may have zero or more databases
- A database may have zero or more ‘collections’.
- A collection may have zero or more ‘documents’.
- A document may have one or more ‘fields’.
- MongoDB ‘Indexes’ function much like their RDBMS counterparts.

RDB Concepts to NO SQL

RDBMS		MongoDB
Database	⇒	Database
Table, View	⇒	Collection
Row	⇒	Document (JSON, BSON)
Column	⇒	Field
Index	⇒	Index
Join	⇒	Embedded Document
Foreign Key	⇒	Reference
Partition	⇒	Shard

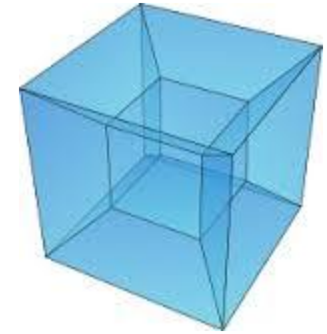
Collection is not strict about what it Stores

Schema-less

Hierarchy is evident in the design

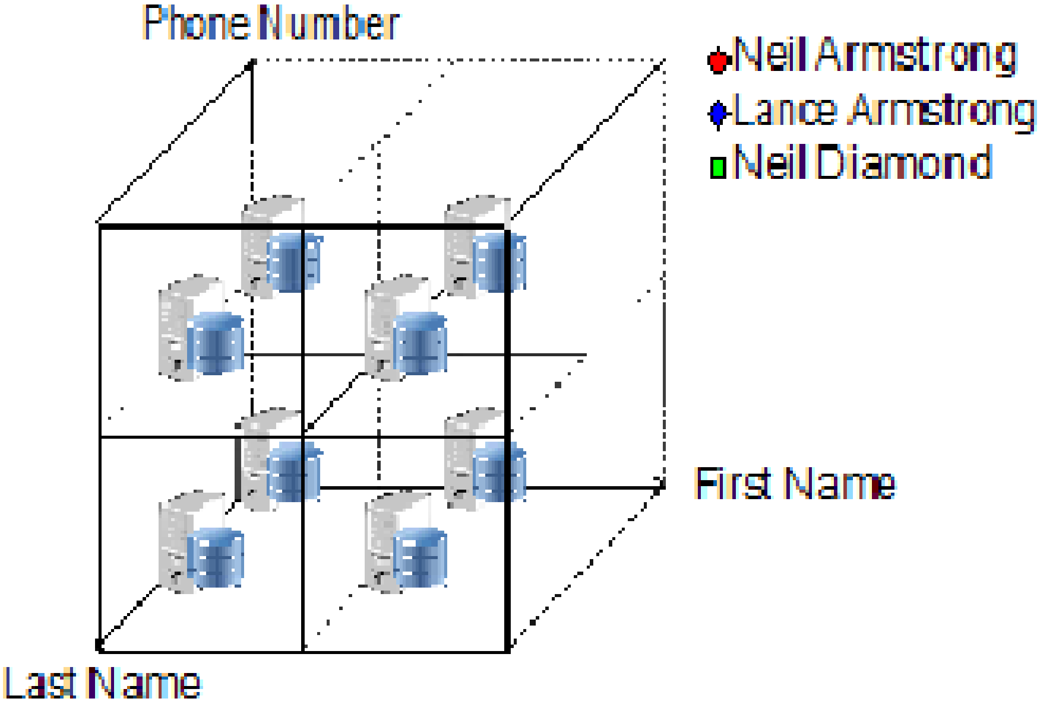
Embedded Document ?

HyperDex Key Points



- Maps records to a Hypercube Space
  - object’s key are stored in a dedicated one-dimensional subspace for efficient lookup
  - only need to contact the servers which match the regions of the hyperspace assigned for the search attributes
- Value-dependent chaining
  - Keeps replicas consistent without heavy overhead from coordination of servers
    - Uses the hypercube space
  - Appoints a point leader that contains the most recent update of a record
    - Other replicas are updated from the point leader

Each server is responsible for a region of the hyperspace



That’s it

- Go over the lecture notes
- Read the book
- Go over homework 3
  - final exam questions will not be as difficult as homework problems
- Ask questions in piazza or via email
- Organize a study sheet
- Complete the example mid-term
- Practice problems

Summary: RAID Levels

- Level 0: No redundancy
- Level 1: Mirrored (two identical copies)
  - Each disk has a mirror image (check disk)
  - Parallel reads, a write involves two disks.
  - Maximum transfer rate = transfer rate of one disk
- Level 0+1: Striping and Mirroring
  - Parallel reads, a write involves two disks.
  - Maximum transfer rate = aggregate bandwidth

Summary: RAID Levels (Contd.)

- Level 3: Bit-Interleaved Parity
  - Striping Unit: One bit. One check disk.
  - Each read and write request involves all disks; disk array can process one request at a time.
- Level 4: Block-Interleaved Parity
  - Striping Unit: One disk block. One check disk.
  - Parallel reads possible for small requests, large requests can utilize full bandwidth
  - Writes involve modified block and check disk
- Level 5: Block-Interleaved Distributed Parity
  - Similar to RAID Level 4, but parity blocks are distributed over all disks