Final Exam Review

Kathleen Durant PhD
CS 3200 Northeastern University

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

- Thursday, December 17, 2015 Room: Cargill Hall 097
- Open books and open notes
  - But no portable devices (no laptops, no phones, etc.)
- 2 hour time period 8:00 to 10:00 AM

Lectures for the final exam

- 9 lectures – starting with October 29's lectures
- **All lectures included**

Text chapters for the final exam

- Transactions: Chapter 16-18
  16. Overview of transaction management
  17. Concurrency control
  18. Recoverability
- Chapters 10-11
  10. Tree-structured index
  11. Hash-based index
- Chapters 12 - 14
  12. Query Evaluation
  13. External sorting
  14. Evaluating Relational Operators
- Chapter 21
  - System Administration (Views)

Topics for the final exam

Topics

- Transactions
  - Serializability
  - 2 Phase Locking
  - Isolation Levels
- Buffer management
  - In relationship to the data manager
- Indexes
  - Primary vs. Secondary
  - Clustered vs. Unclustered
  - Tree-structured: ISAM, B+ trees
  - Hash-based indexes
- System Administration - Views
- Query Evaluation
- Query Optimization
- NO SQL

Algorithms

- Insertion/Deletion of records
  - B+ tree Index
  - ISAM
  - Extendible hashing index

Format of the final exam

- 1-2 Algorithmic/Calculation problems (40%)
  - 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
  - Serializability
- Some close-ended responses (30%)
  - Short collection of True and False
  - Multiple choice
  - Short definitions

Study Steps

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

CONTENT REVIEW

What is a transaction?

- A transaction is a collection of operations treated as a single logical operation
  - Typically carried out by a single user or an application program
  - Reads or updates the contents of a database
- A transaction is a ‘logical unit of work’ on a database
  - Each transaction does something in the database
  - No part of it alone achieves anything of use or interest to a user
- Transactions are the unit of recovery, consistency, and integrity of a database
- A *transaction* is the DBMS’s abstract view of a user program: a sequence of reads and writes.

Transactions: ACID Properties

- **Atomicity**: either the entire set of operations happens or none of it does
- **Consistency**: the set of operations taken together should move the system for one consistent state to another consistent state.
- **Isolation**: each system perceives the system as if no other transactions were running concurrently (even though odds are there are other active transactions)
- **Durability**: results of a completed transaction must be permanent - even IF the system crashes

Concurrency Control

Process of managing simultaneous operations on the database without having them interfere with one another.

- **Prevents interference when two or more users are accessing the database simultaneously and at least one is updating data.**
- **Although two transactions may be correct in themselves, interleaving of operations may produce an incorrect result.**

Serializability

Schedule

Sequence of reads/writes by set of concurrent transactions.

Serial Schedule

Schedule where operations of each transaction are executed consecutively without any interleaved operations from other transactions.

- No guarantee that results of all serial executions of a given set of transactions will be identical.

Serial**izable** schedule: alternative to simple serial schedule

- Multiple transactions running: we know that the execution of a set of simultaneous transactions is correct if it obeys the ACID properties
- More formally:
  - Define the sequence of operations performed is a schedule.
  - Define the sequence of operations performed when running each transaction serially as a serial schedule.
  - **Any schedule that *corresponds* to a serial schedule is correct.**

Nonserial Schedule

- **Schedule where operations from set of concurrent transactions are interleaved.**
- **Objective of serializability is to find nonserial schedules that allow transactions to execute concurrently without interfering with one another.**
- **In other words, want to find nonserial schedules that are equivalent to *some* serial schedule. Such a schedule is called *serializable*.**

15

Serializability

- **In serializability, ordering of read/writes is important:**

  (a) If two transactions only read a data item, they do not conflict and order is not important.

  (b) If two transactions either read or write separate data items, they do not conflict and order is not important.

  (c) If one transaction writes a data item and another reads or writes same data item, order of execution is important.

Database Recovery

Process of restoring database to a correct state in the event of a failure.

- **Need for Recovery Control**
  - **Two types of storage: volatile (main memory) and nonvolatile.**
  - **Volatile storage does not survive system crashes.**
  - **Stable storage represents information that has been replicated in several nonvolatile storage media with independent failure modes.**

Types of Failures

- System crashes, resulting in loss of main memory.
- Media failures, resulting in loss of parts of secondary storage.
- Application software errors.
- Natural physical disasters.
- Carelessness or unintentional destruction of data or facilities.
- Sabotage.

Transactions and Recovery

- **Transactions represent basic unit of recovery.**
- **Recovery manager responsible for atomicity and durability.**
- **If failure occurs between commit and database buffers being flushed to secondary storage then, to ensure durability, recovery manager has to *redo* (*rollforward*) transaction's updates.**

Transactions and Recovery

- **If transaction had not committed at failure time, recovery manager has to *undo* (*rollback*) any effects of that transaction for atomicity.**
- **Partial undo - only one transaction has to be undone.**
- **Global undo - all transactions have to be undone.**

Buffer pool management

- FORCE – every write to disk?
  - Poor performance (many writes clustered on same page)
  - At least this guarantees the persistence of the data
- STEAL – allow dirty pages to be written to disk?
  - If so, reading data from uncommitted transactions violates atomicity
  - If not, poor performance

	Force - every write to disk	**No Force – write when optimal**
Steal – use internal DB buffer for read		Desired but complicated
No Steal - always read only committed data	Easy but slow	

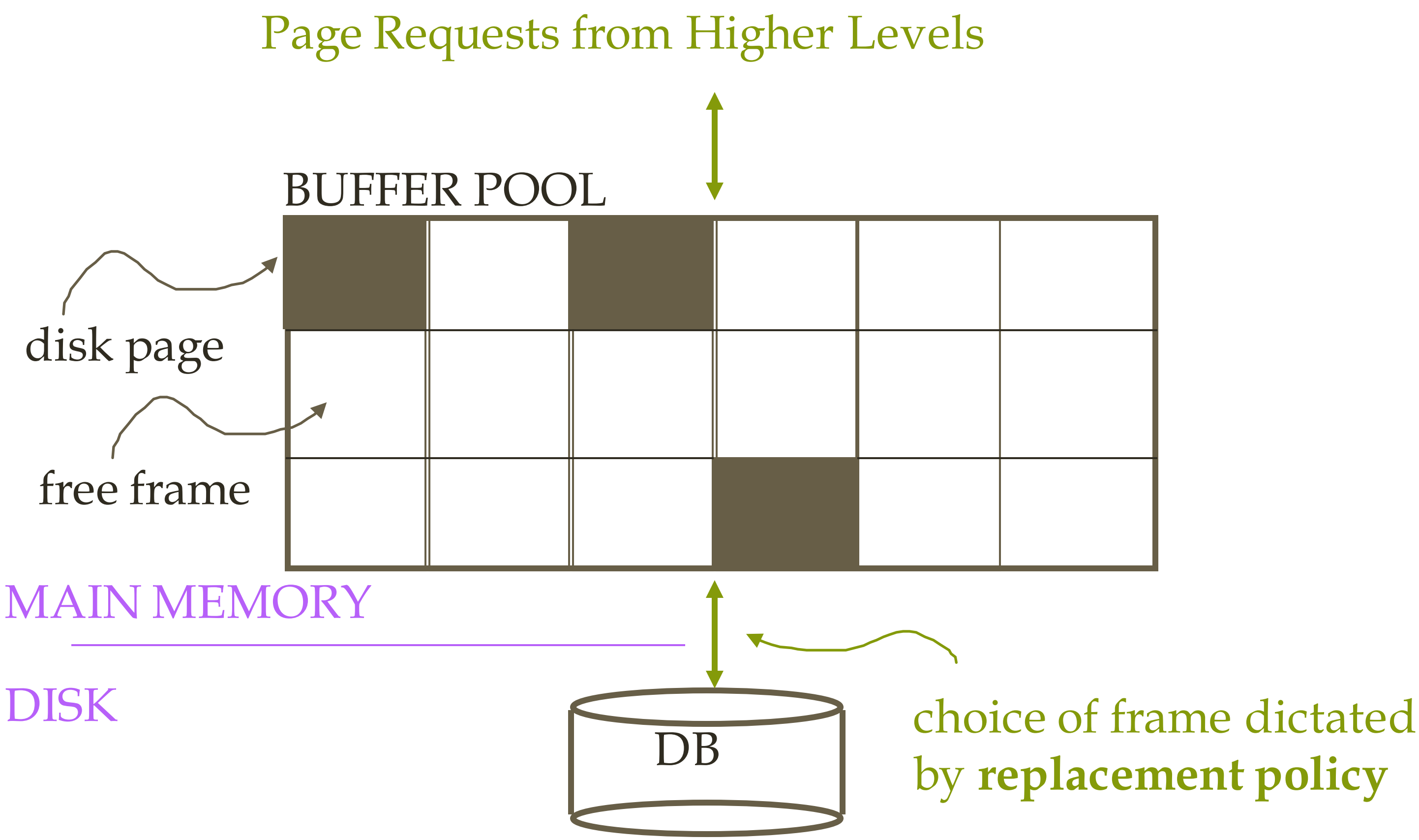
Complications from NO FORCE and STEAL

- NO FORCE
  - What if the system crashes before a modified page can be written to disk?
  - Write as little as possible to a convenient place at commit time to support **REDO**ing the data update
- STEAL
  - Current updated data can be flushed to disk but still locked by a transaction T1
    - What if T1 aborts?
    - Need to **UNDO** the data update done by T1

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.

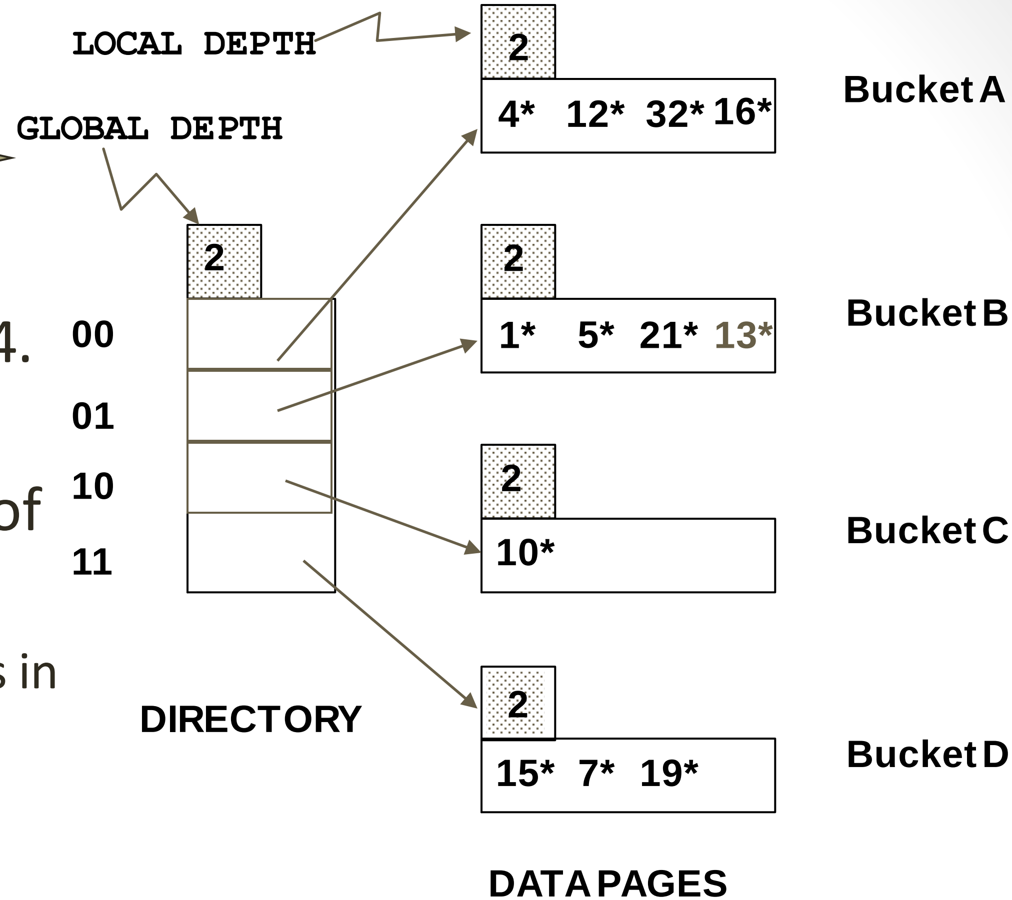
Index classification

- 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.

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!

Extendible Hashing Example

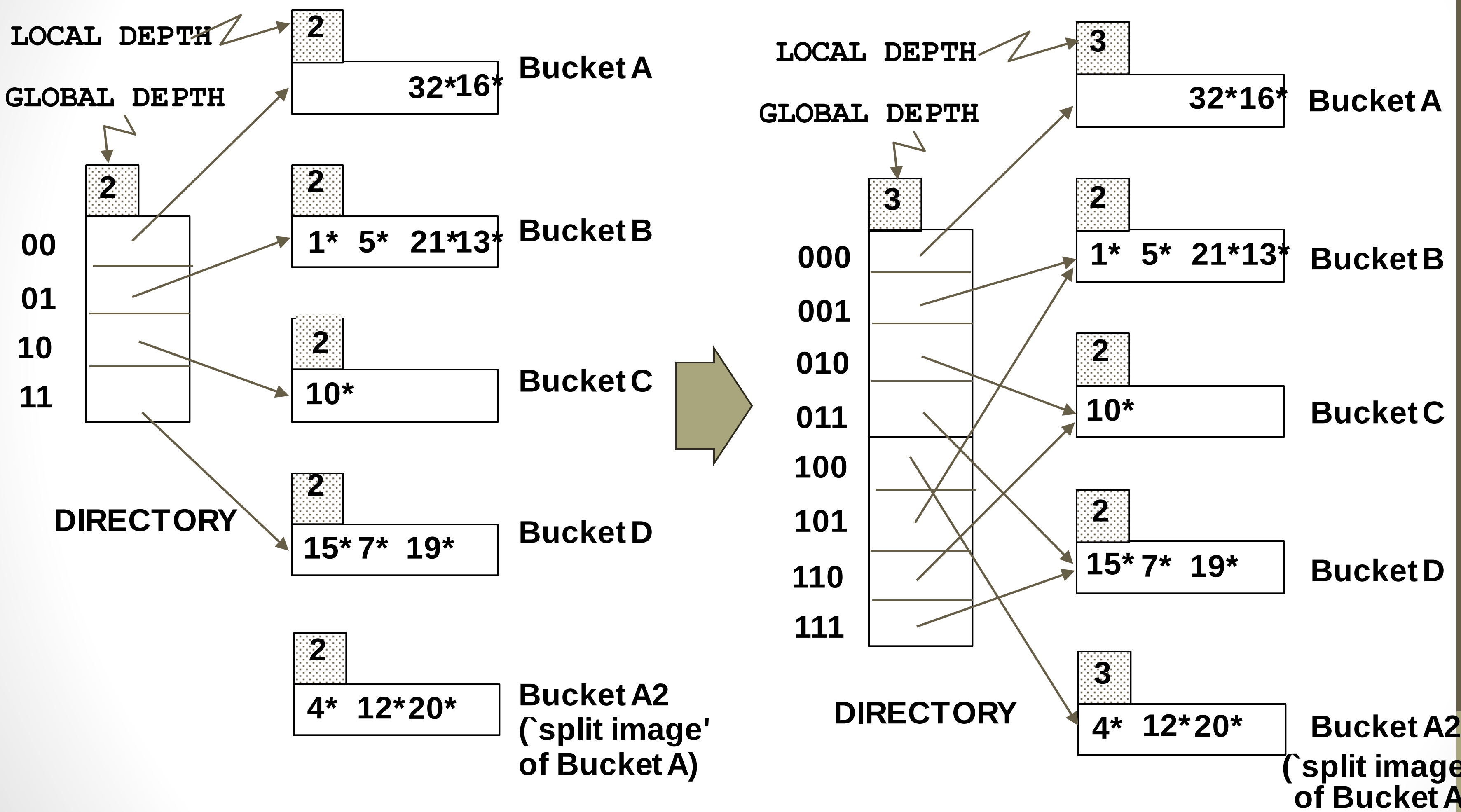


- Directory is array of size 4.
- To find bucket for *r*, take last \`*global depth*' # bits of **h**(*r*); we denote *r* by **h**(*r*).
  - If **h**(*r*) = 5 = binary 101, it is in bucket pointed to by 01.

❖ **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!)

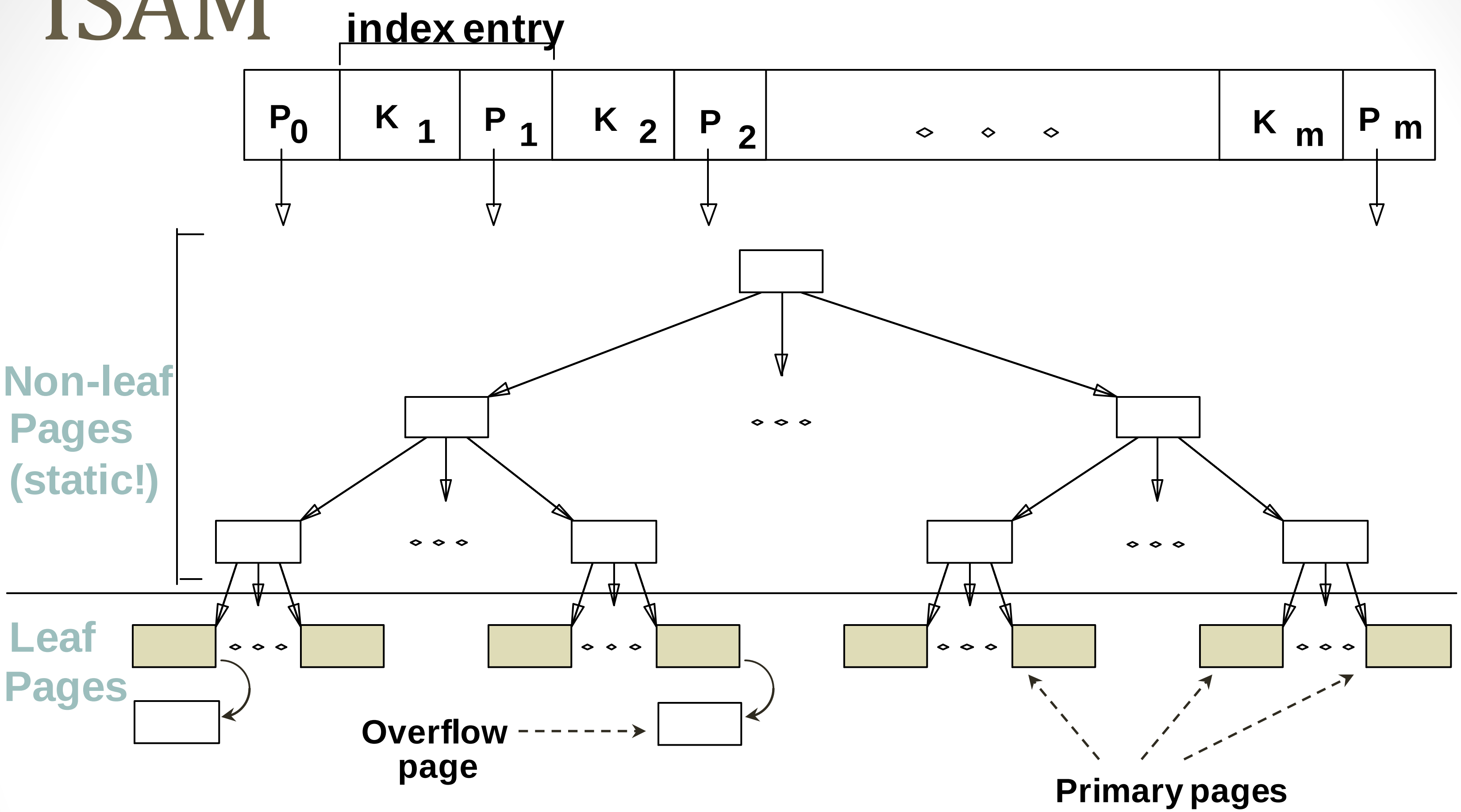
Summary: Hash-Based Indexes

- 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.

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

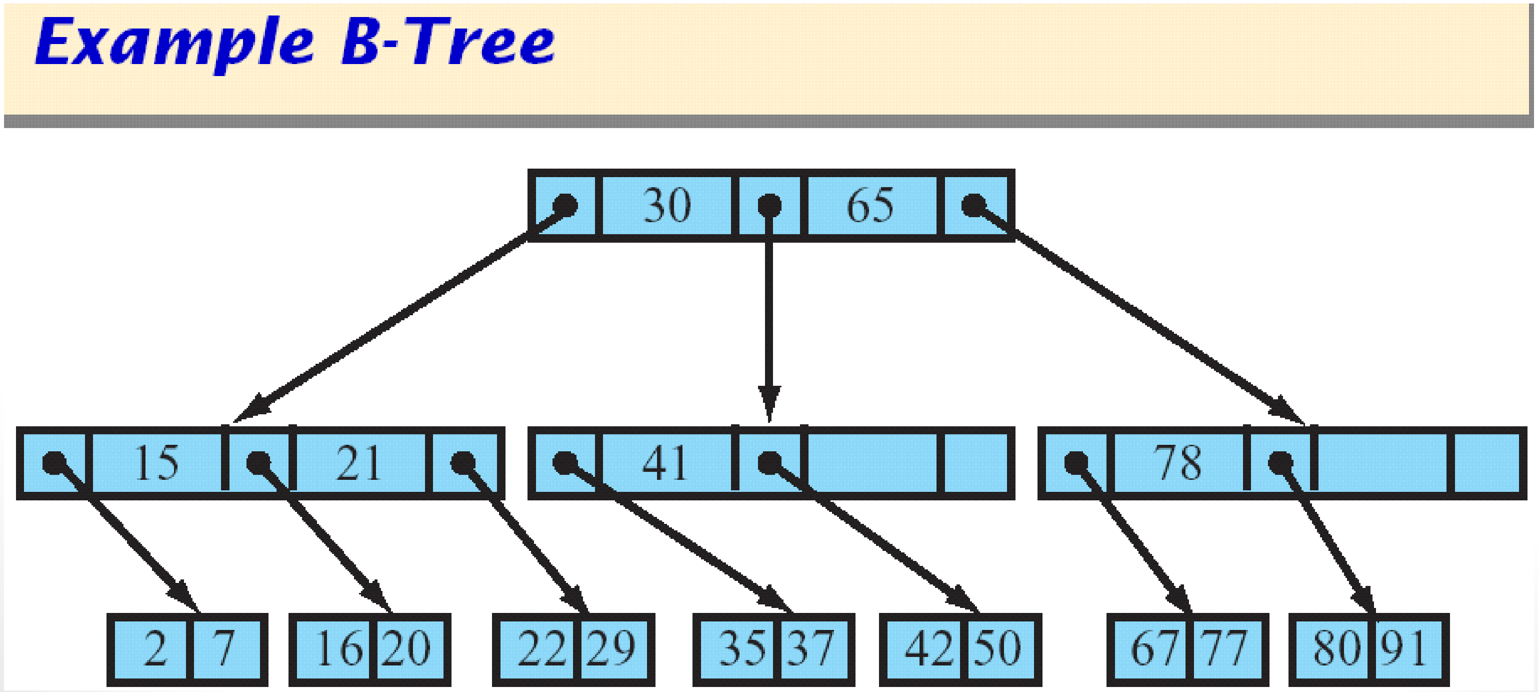
A B-tree helps minimize access to the index / directory

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

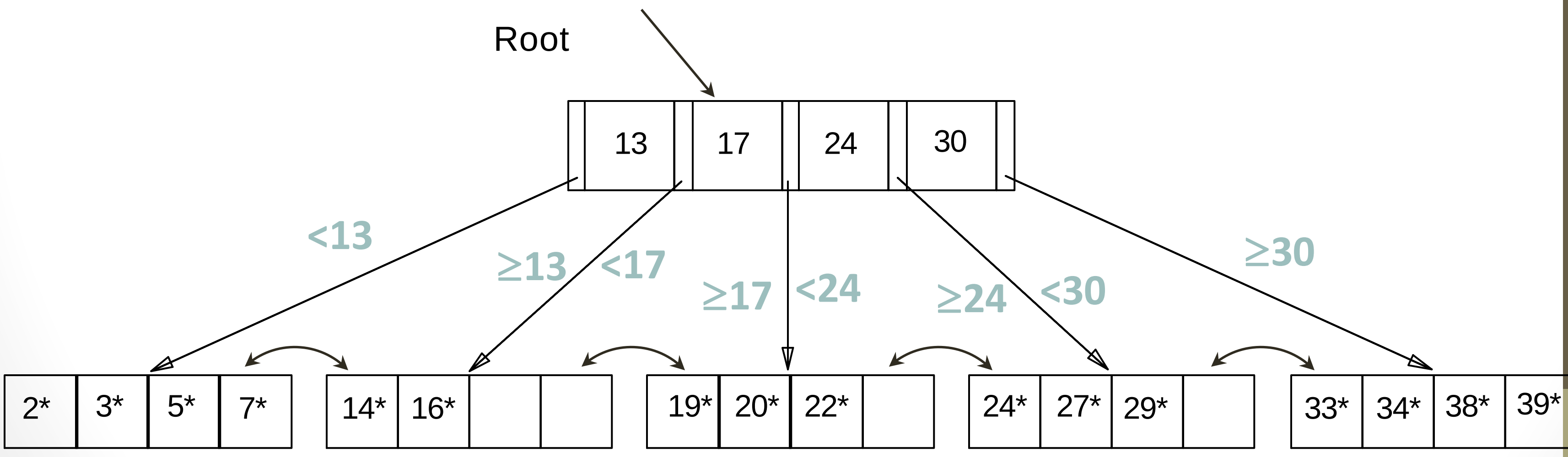


Definition of B+ Tree

- 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 $\lceil$ **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.

QUERY EVALUATION AND QUERY OPTIMIZATION

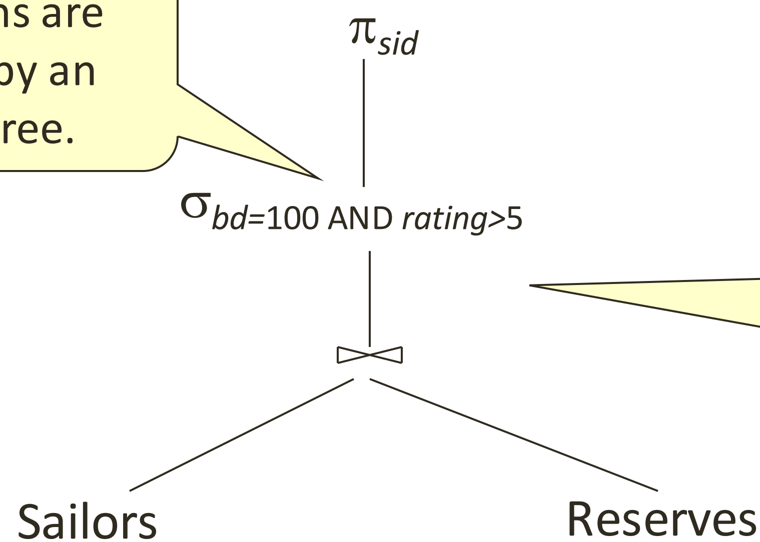
Tree of relational operators

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;
```

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

RA expressions are represented by an expression tree.



An algorithm is chosen for each node in the expression tree.

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* ($\bigcup$) 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*.

JOIN Algorithms

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

Select functionality

Influences the use of sorting and hashing

Project functionality

- General selection criteria
- Answering queries via record ids

Join Algorithms R JOIN S

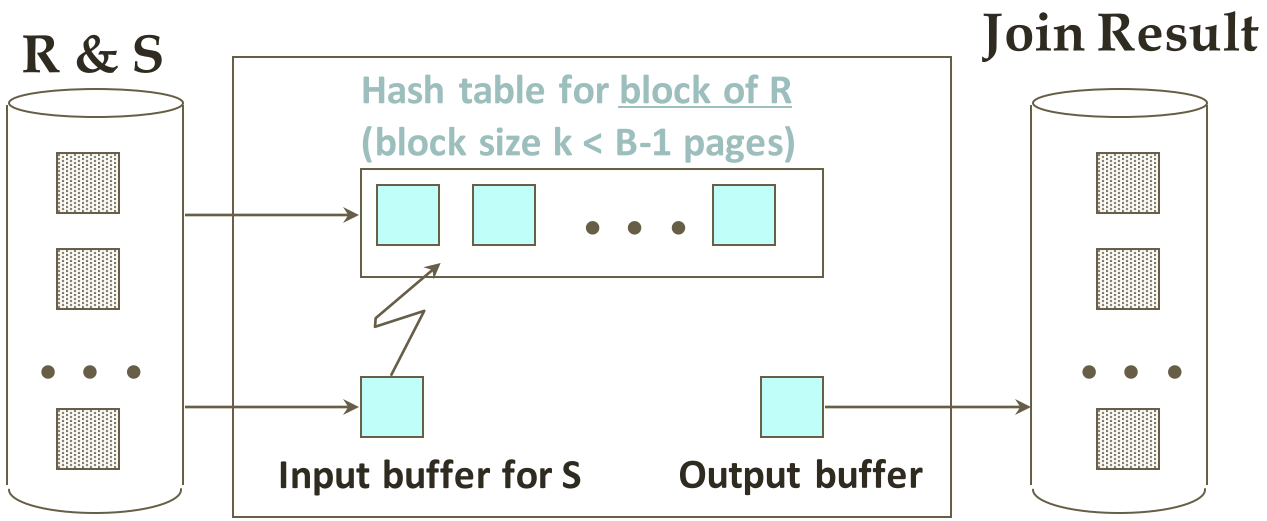
Algorithm	Cost
Block Nested Loop Join	Nblocks(R) + (nBlocks(R)*nblocks(S)), if buffer has 1 block for R and S nBlocks(R) +[nblocks(S)*nblocks(R)/(nbuffer – 2))] if (nBuffer – 2) blocks for R nBlocks(R) + nBlocks(S)m if all blocks of R can be read into a database buffer
Index Nested Loop Join	Depends on Indexing method: nBlocks(R) + ntuples(R)*(nLevels(I) + 1) if join attribute A in S is the primary key nBlocks(R) + ntuples(R)*(nLevels $_A$(I) + [SC$_A$ (R)/bFactor(R)]) for secondary clustering (SC) index I on attribute A
Sort-merge Join	Nblocks(R) *[log$_2$(nblocks(R)] + nBlocks(S)*[log$_2$ (nblocks(S)], for sort Nblocks(R) + nblocks(S) for merge
Hash Join	3(nBlocks(R) + nblocks(S)), if hash index is held in memory 2(nblocks(r) + nblocks(S))*[log $_{nbuffer-1}$(nblocks(S)) – I] + nblocks(R) + nblocks(S), otherwise

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 = ⌈ # pages of outer / block size ⌉
  - 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

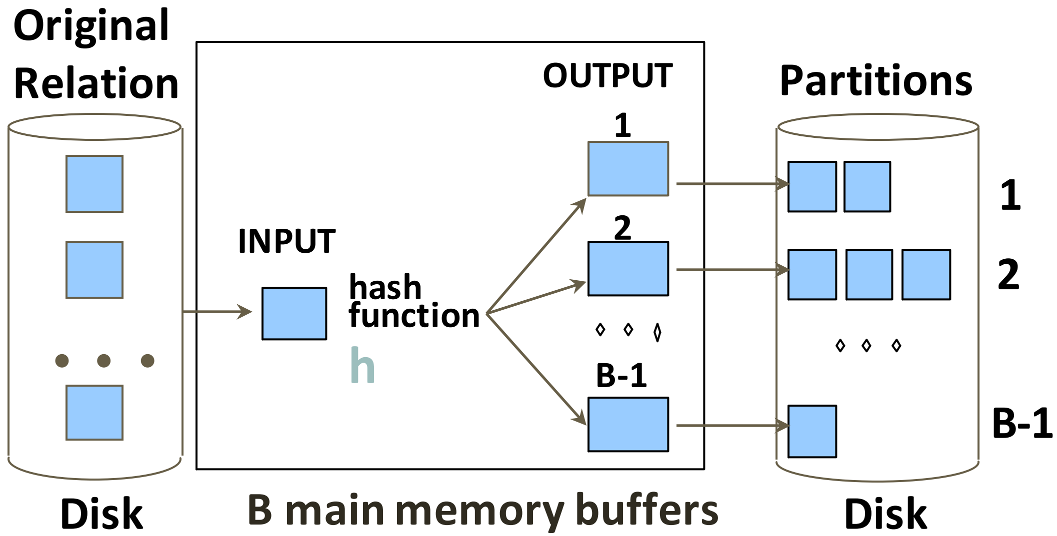
Refinement of Sort-Merge Join

- *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.

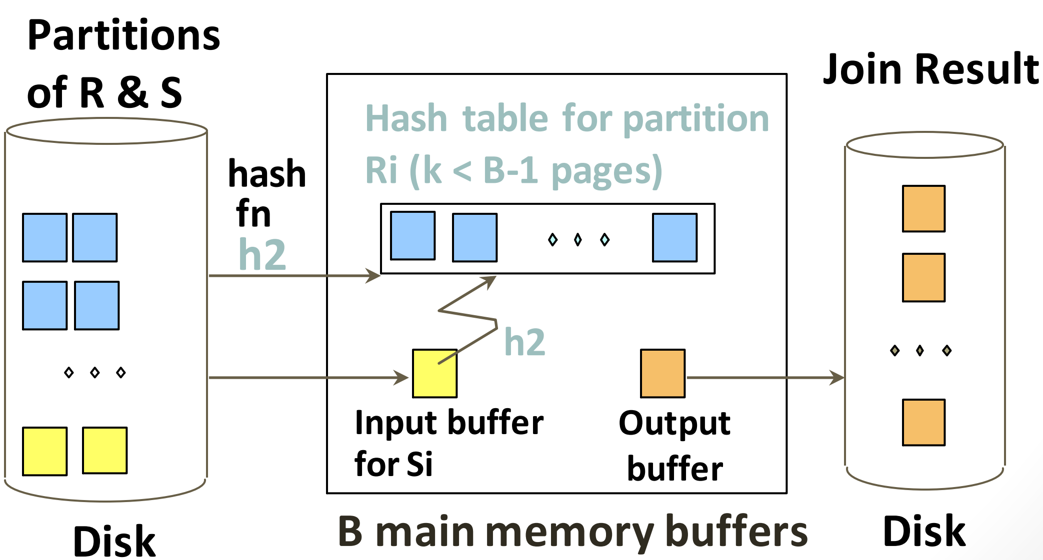
Hash Join

- *Idea*: Partition both R and S using a hash function s.t. R tuples will only match S tuples in partition i.

- 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 :
  - 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*, we can:
    - Retrieve rids of records satisfying *day<8/9/94* using the first index
    - Retrieve rids of records satisfying *sid=3* using the other index
    - intersect these rids
    - retrieve records and check *bid=5.*

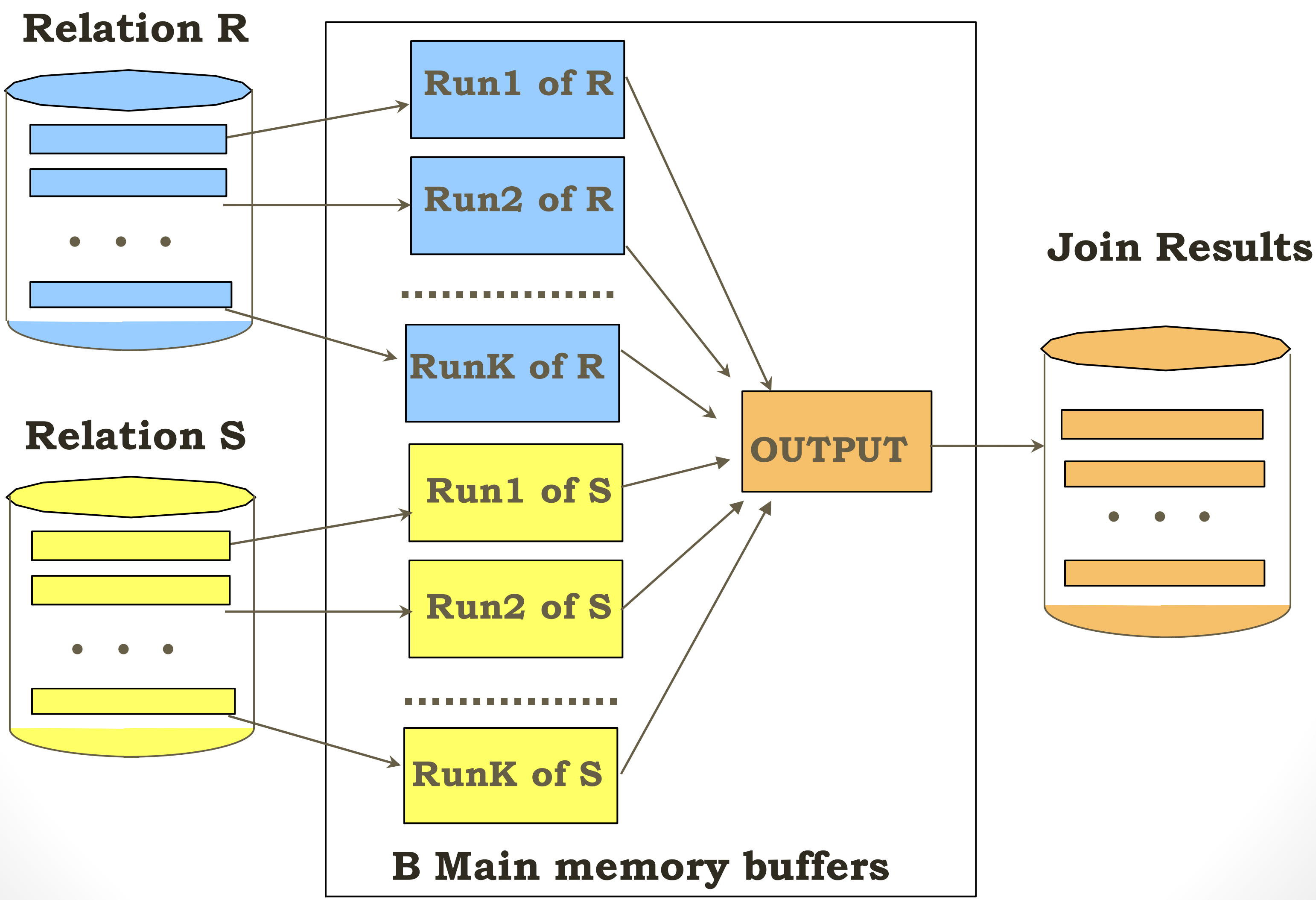
Projection Based on Sorting

- 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.

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 secondary indexes:
  - 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.

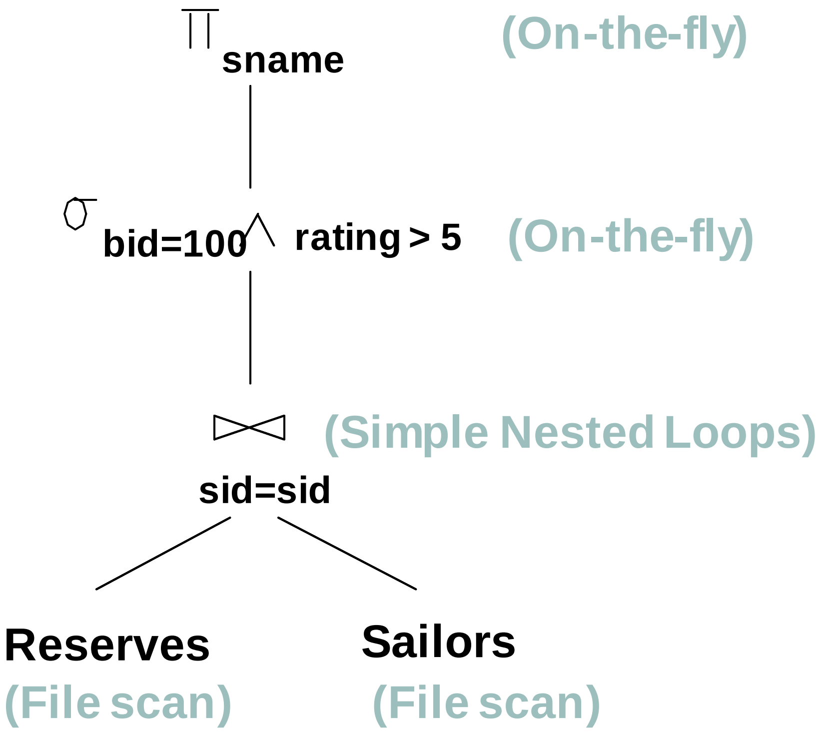
QUERY EVALUATION PLAN

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 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'.

VIEWS

Views: another data limitation mechanism

<u>View</u>

Dynamic result of one or more relational operations operating on base relations to produce another relation.

- **Virtual relation that does not necessarily actually exist in the database but is produced upon request, at time of request.**

Views

- **Contents of a view are defined as a query on one or more base relations.**
- **With view resolution, any operations on the view are automatically translated into operations on relations from which it is derived.**
- **With view materialization, the view is stored as a temporary table, which is maintained as the underlying base tables are updated.**

Some benefits provided by views

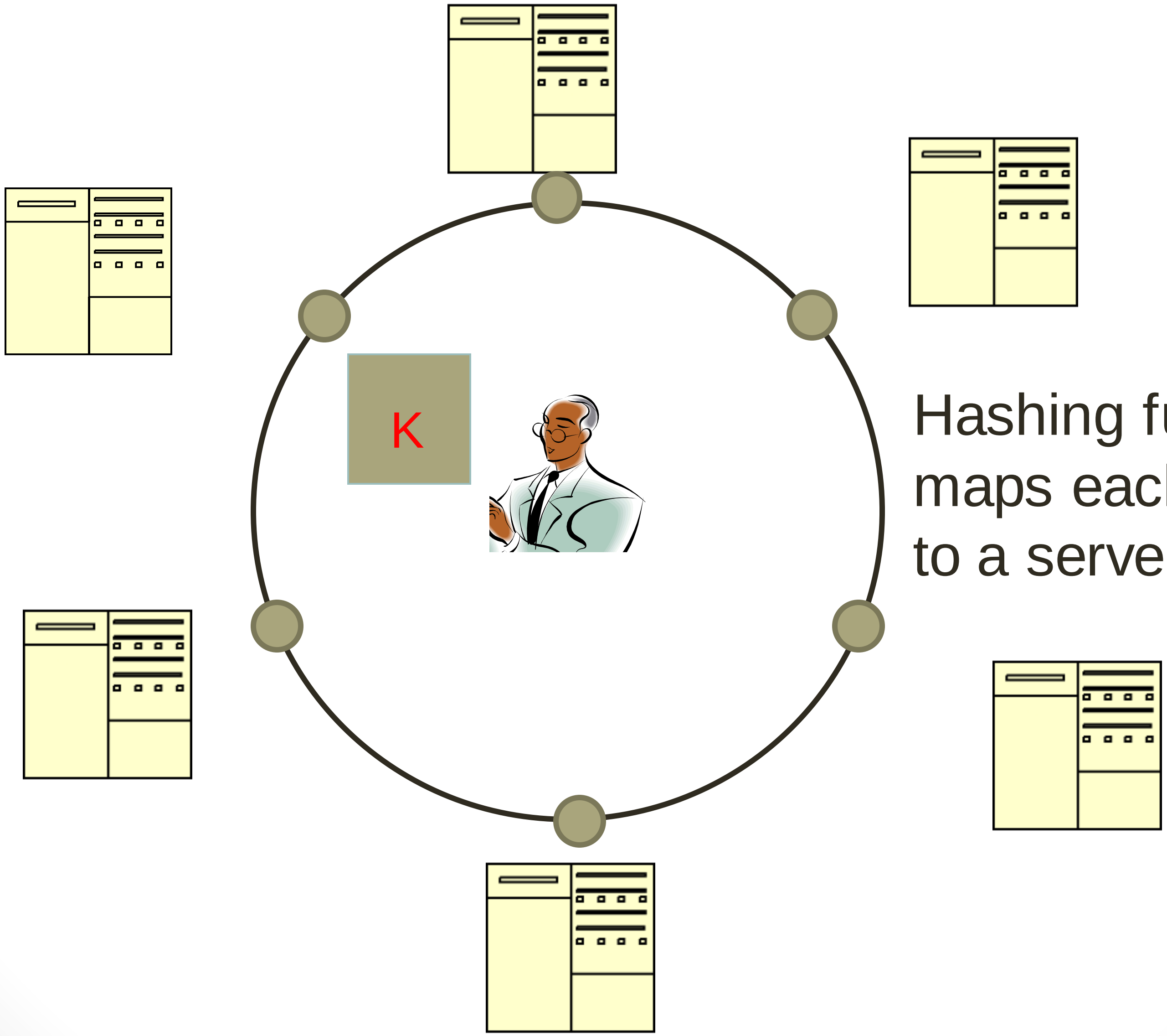
- **Data independence**
- **Improved security**
- **Reduced complexity**
- **Convenience**
- **Customization**
- **Data integrity**
- **Concurrency**

Disadvantages of Views

- **Update restriction**
- **Structure restriction**
- **Performance**

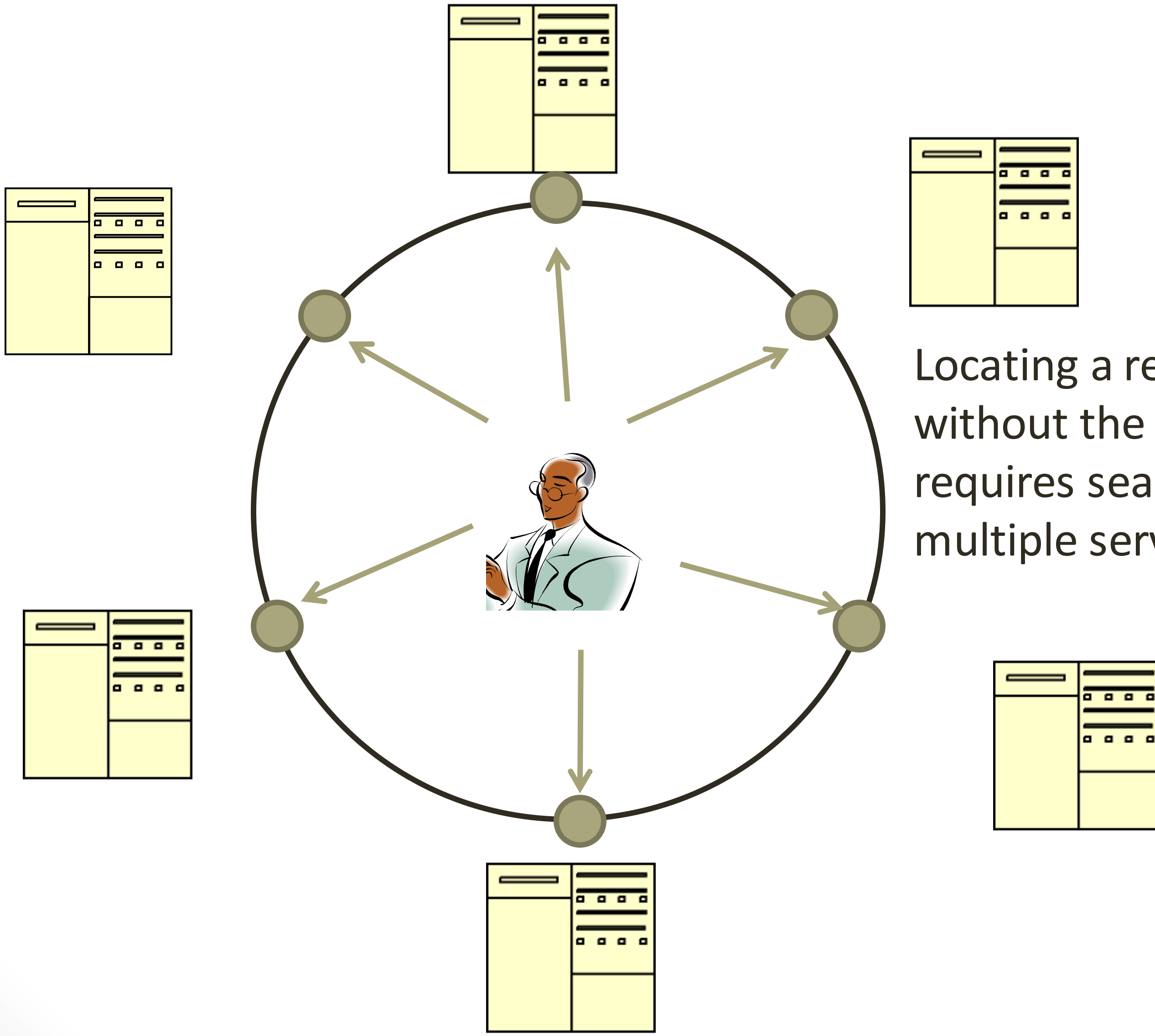
NO SQL

Typical NoSQL architecture



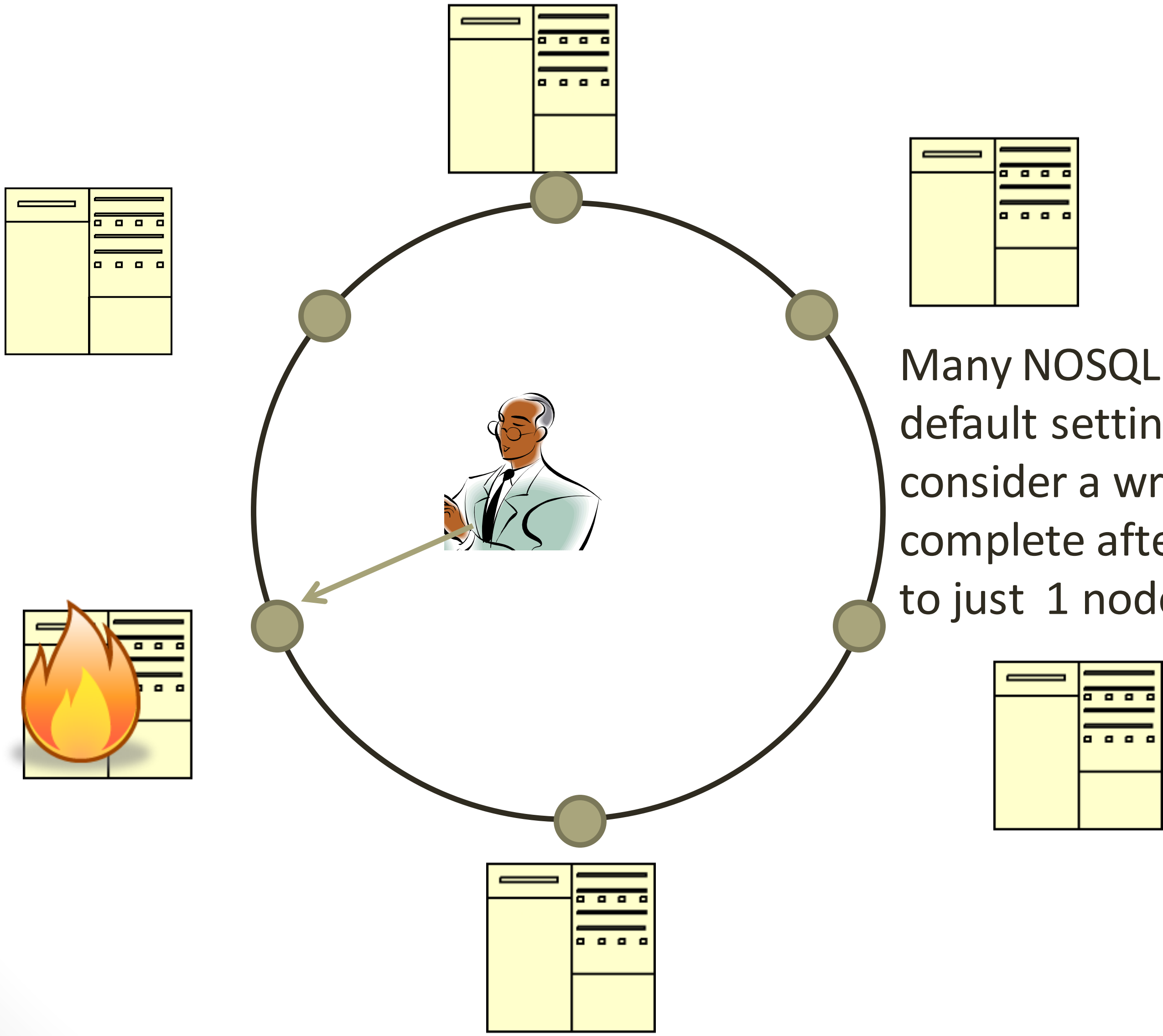
Hashing function maps each key to a server

The search problem: No Hash key



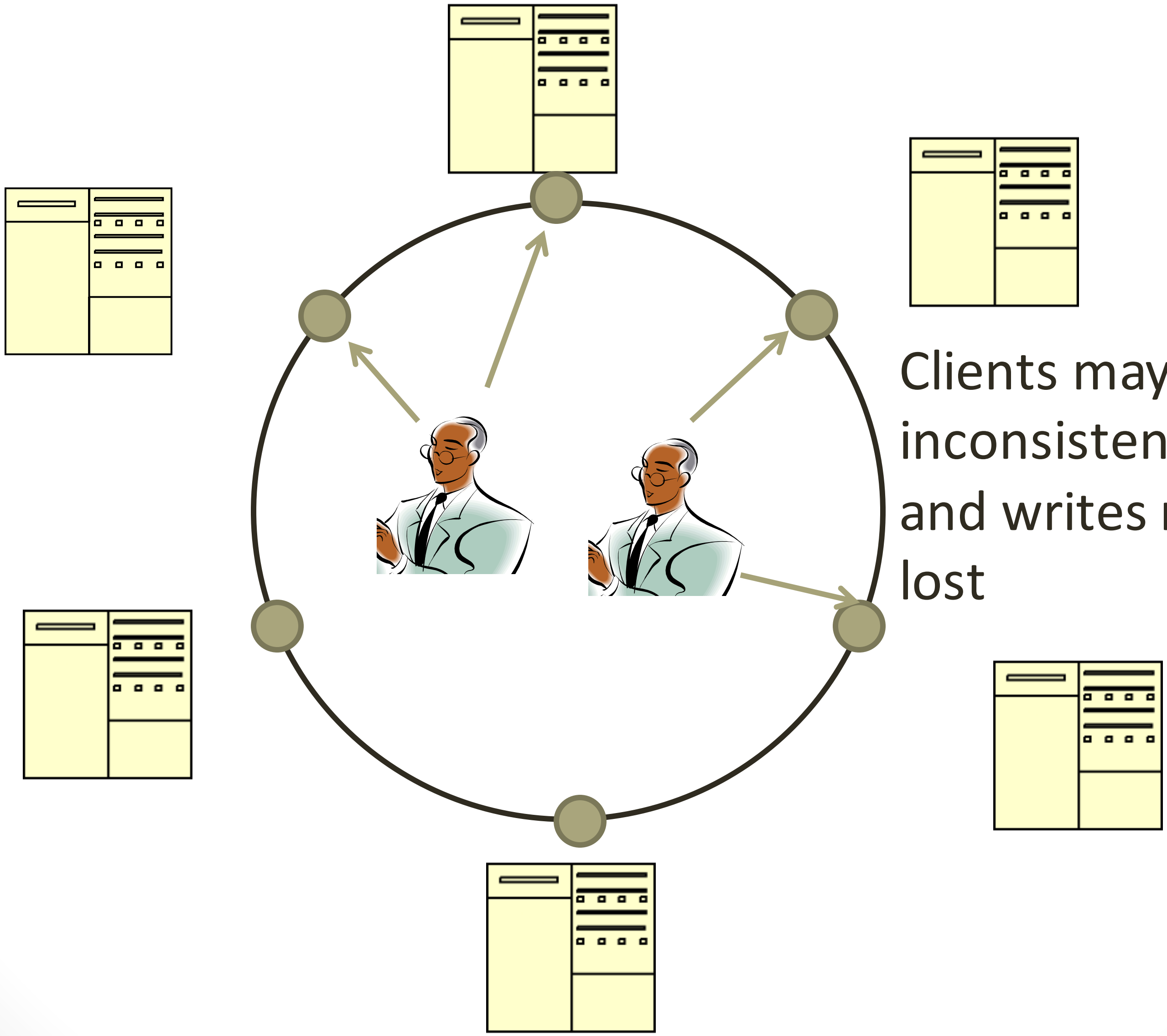
Locating a record without the hash key requires searching multiple servers

The Fault Tolerance problem



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

The consistency problem



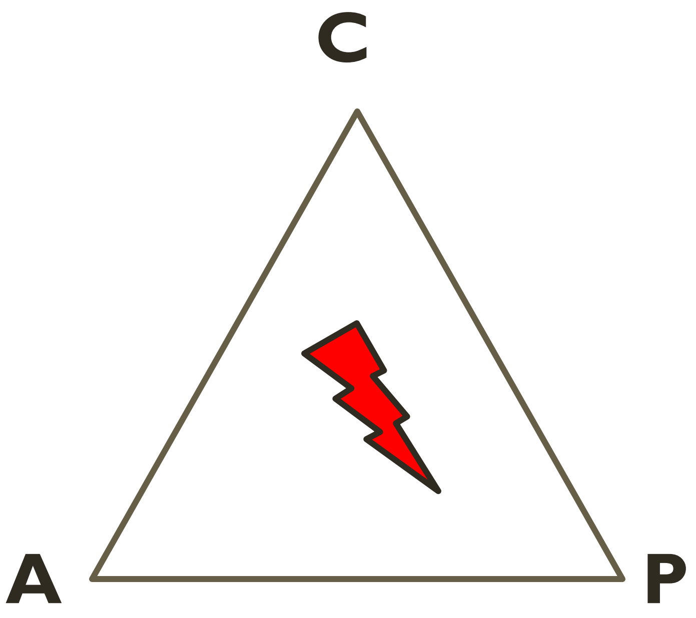
Clients may read inconsistent data and writes may be lost

Theory of NOSQL: CAP

GIVEN:

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

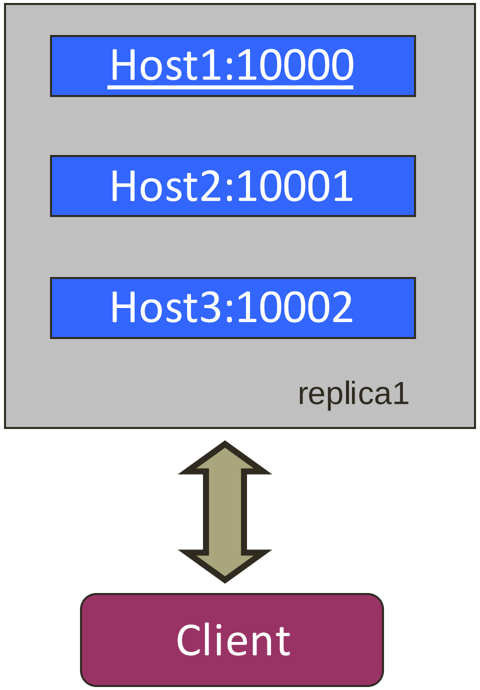
- **Consistency**
  - all replicas contain the same version of data
- **Availability**
  - system remains operational on failing nodes
- **Partition tolarence**
  - multiple entry points
  - system remains operational on system split



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

Replica Sets

- Redundancy and Failover
- Zero downtime for upgrades and maintenance



- 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 ?

That's it

- Go over the lecture notes
- Read the book
- Ask questions in piazza or via email
- Organize a study sheet
- Practice problems