CS150A Database

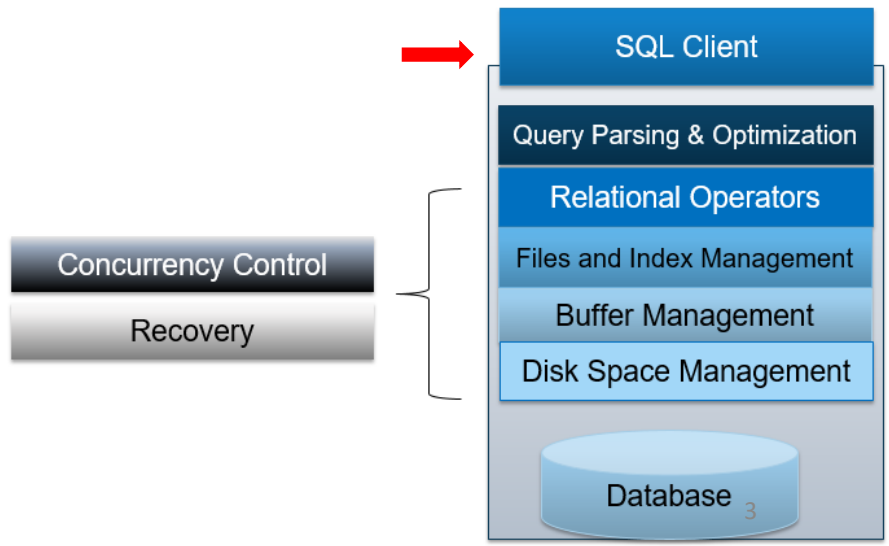
Course Review

Dec. 22, 2022

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

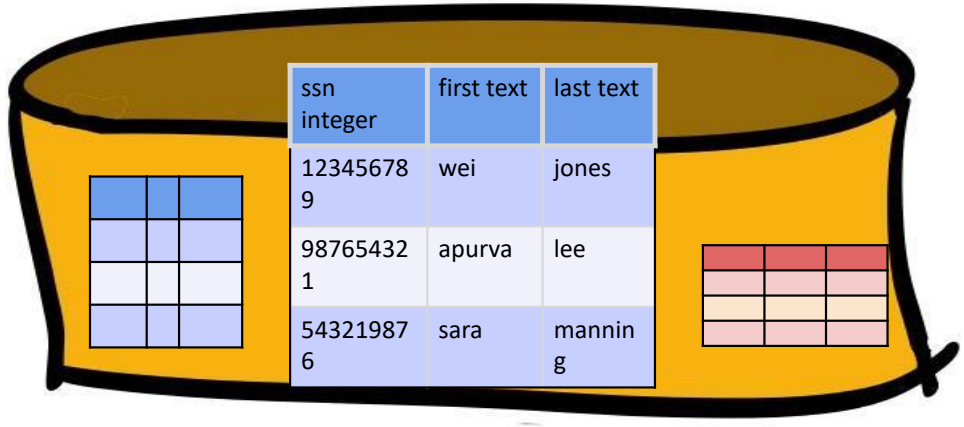
1. SQL


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

Relational Terminology

- ***Database***: Set of named Relations
- ***Relation*** (Table):
  - ***Schema***: description ("metadata")
  - ***Instance***: set of data satisfying the schema
- ***Attribute*** (Column, Field)
- ***Tuple*** (Record, Row)

543219876	sara	manning



SQL Language

- Two sublanguages:
  - DDL – Data Definition Language
    - Define and modify schema
  - DML – Data Manipulation Language
    - Queries can be written intuitively.
- RDBMS responsible for efficient evaluation.
  - Choose and run algorithms for declarative queries
    - Choice of algorithm must not affect query answer.

Declarative!

The SQL DDL: Primary Keys


```
CREATE TABLE Sailors (
        sid INTEGER,
        sname CHAR(20),
        rating INTEGER,
        age FLOAT
```

sid	**sname**	**rating**	**age**
1	Fred	7	22
2	Jim	2	39
3	Nancy	8	27

- Primary Key column(s)
  - Provides a unique “lookup key” for the relation
  - Cannot have any duplicate values
  - Can be made up of >1 column
    - E.g. (firstname, lastname)

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

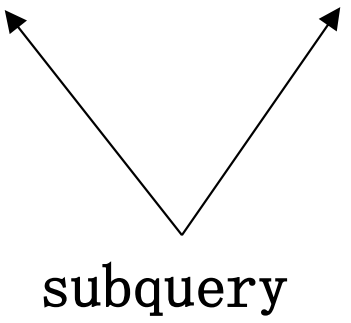
- Subtle connections between:
  - Boolean logic in WHERE (i.e., AND, OR)
  - Traditional Set operations (i.e. INTERSECT, UNION)
    - Set: a collection of distinct elements
    - Standard ways of manipulating/combining sets
      - Union
      - Intersect
      - Except
    - Treat tuples within a relation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



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 …

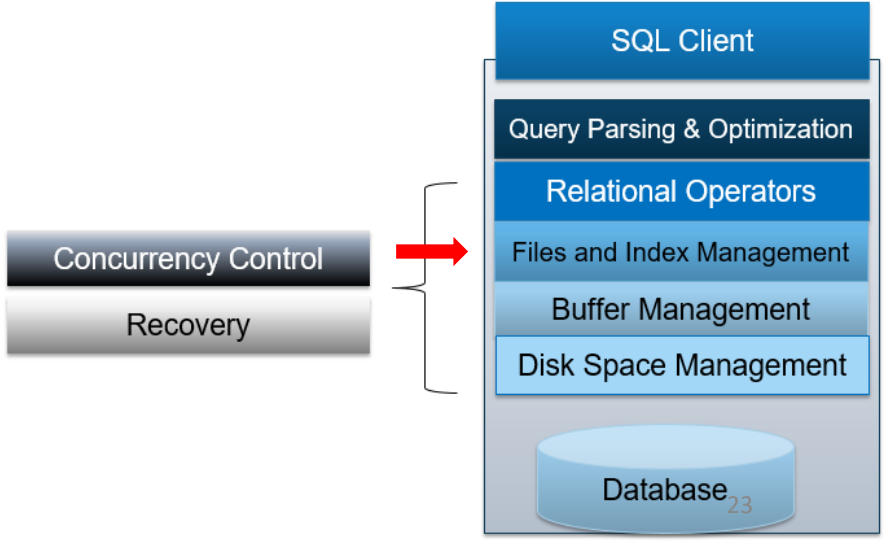
- INNER is default
- Inner join what we've learned so far
  - Same thing, just with different syntax.

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

3. Index and B+ Trees



Index

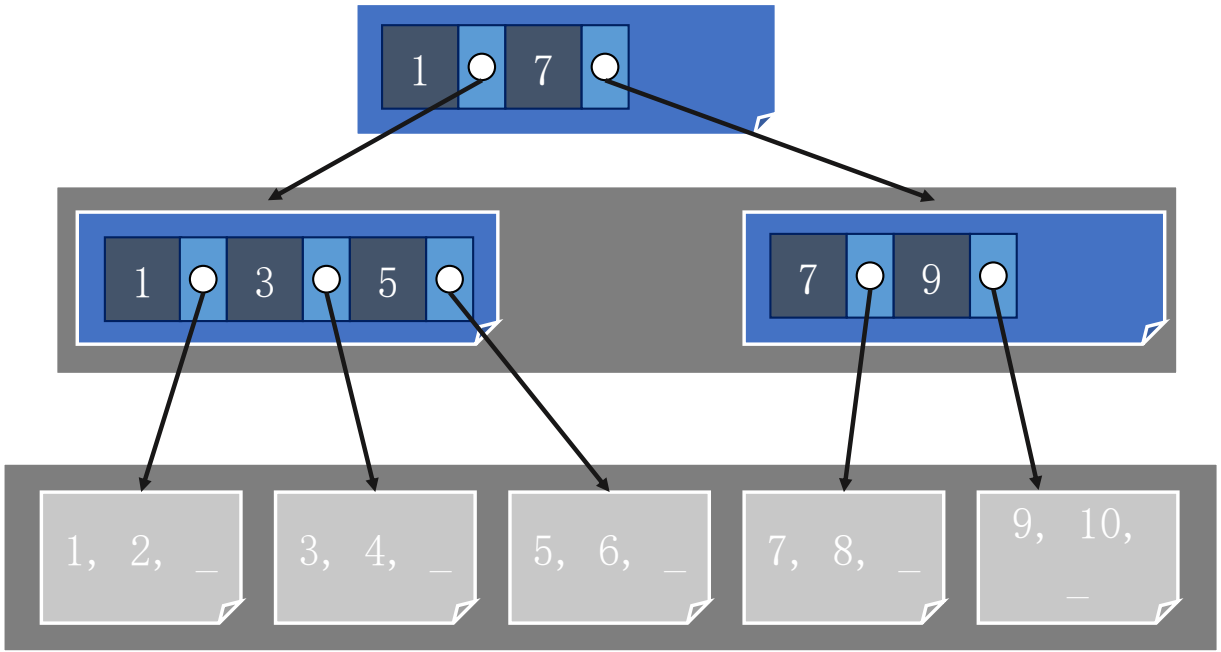
An **index** is data structure that enables fast **lookup** and **modification** of **data entries** by **search key**

- **Lookup**: may support many different operations
  - **Equality**, 1-d range, 2-d region, …
- **Search Key:** any subset of columns in the relation
  - Do not need to be unique
    - —e.g. (firstname) or (firstname, lastname)
- Data Entries: items stored in the index
  - Assume for today: a pair (k, recordId) …
    - Pointers to records in Heap Files!
    - Easy to generalize later
- Modification: want to support fast insert and delete

Many Types of indexes exist: B+-Tree, Hash, R-Tree, GiST, ...

Build a high fan-out search tree Part 4

- Recursively "index" key file
- **Key Invariant:**
  - Node […, (K_L, P_L), (K_R, P_R), ...] → All tuples in range K_L <= K < K_R are in tree P_L



Searching for 5?

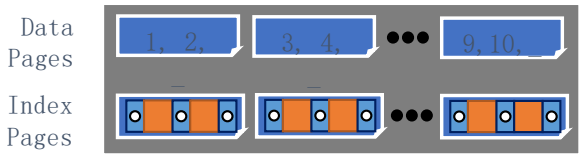
Binary Search each node (page) starting at root

Follow pointers to next level of search tree

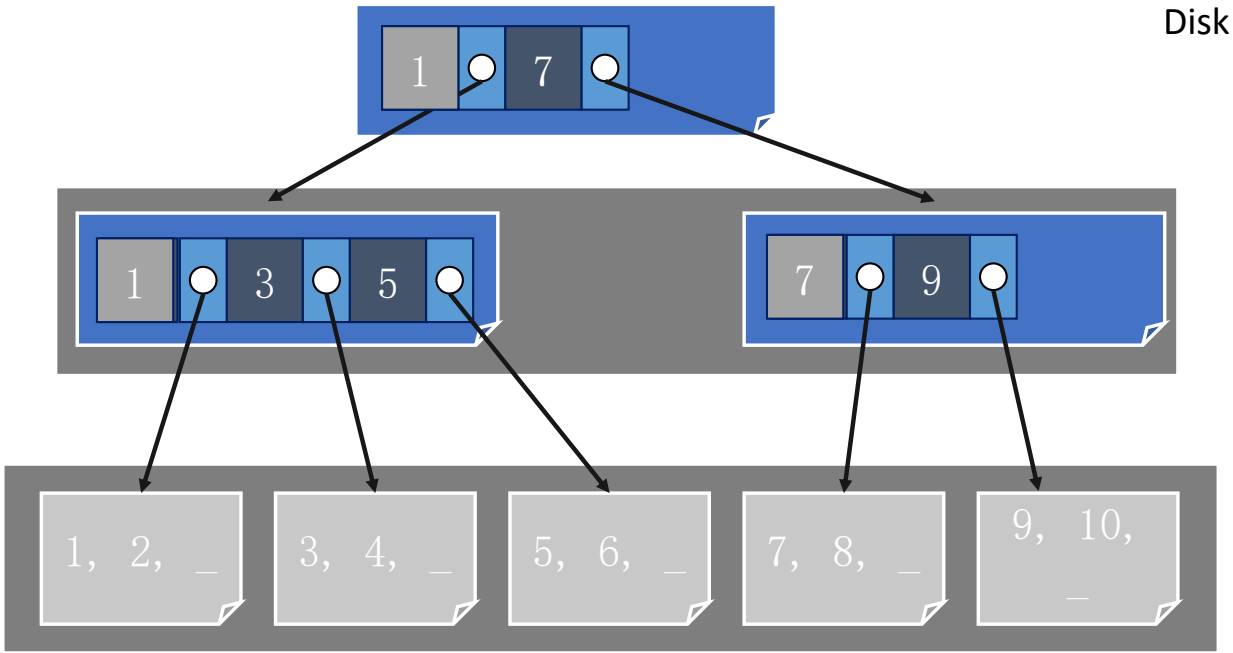
Complexity? $O(\log_F(\#Pages))$

Left Key Optimization?

- Optimization
  - Do we need the left most key?



Disk Layout? All in a single file, Data Pages first.

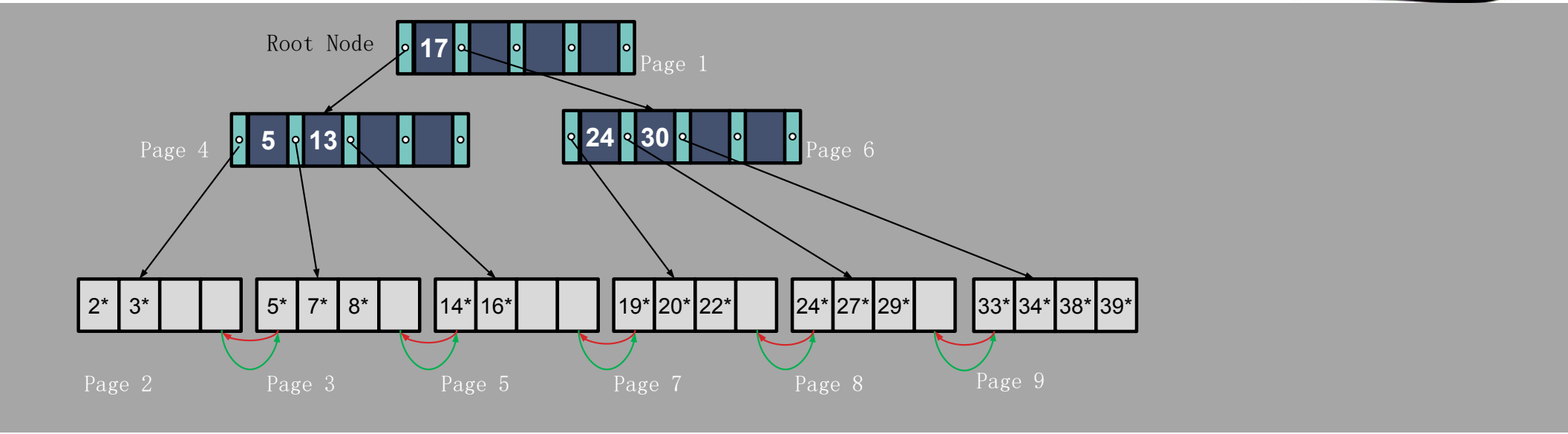


ISAM

Indexed Sequential Access Method

(Early IBM Indexing Technology)

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

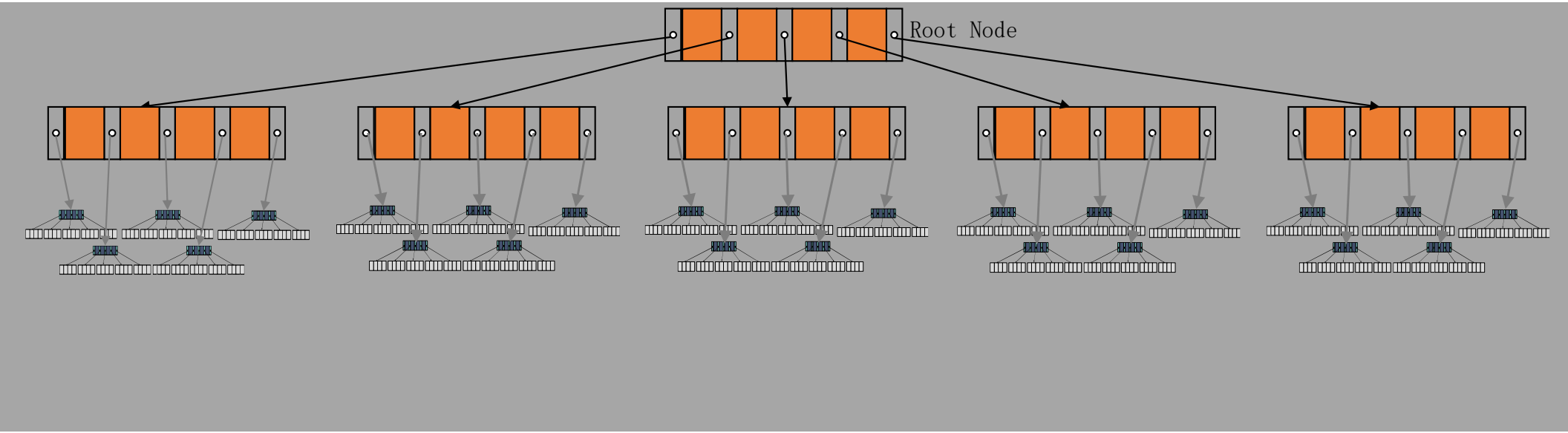
What about the root?

The root is special

Why not in sequential order?

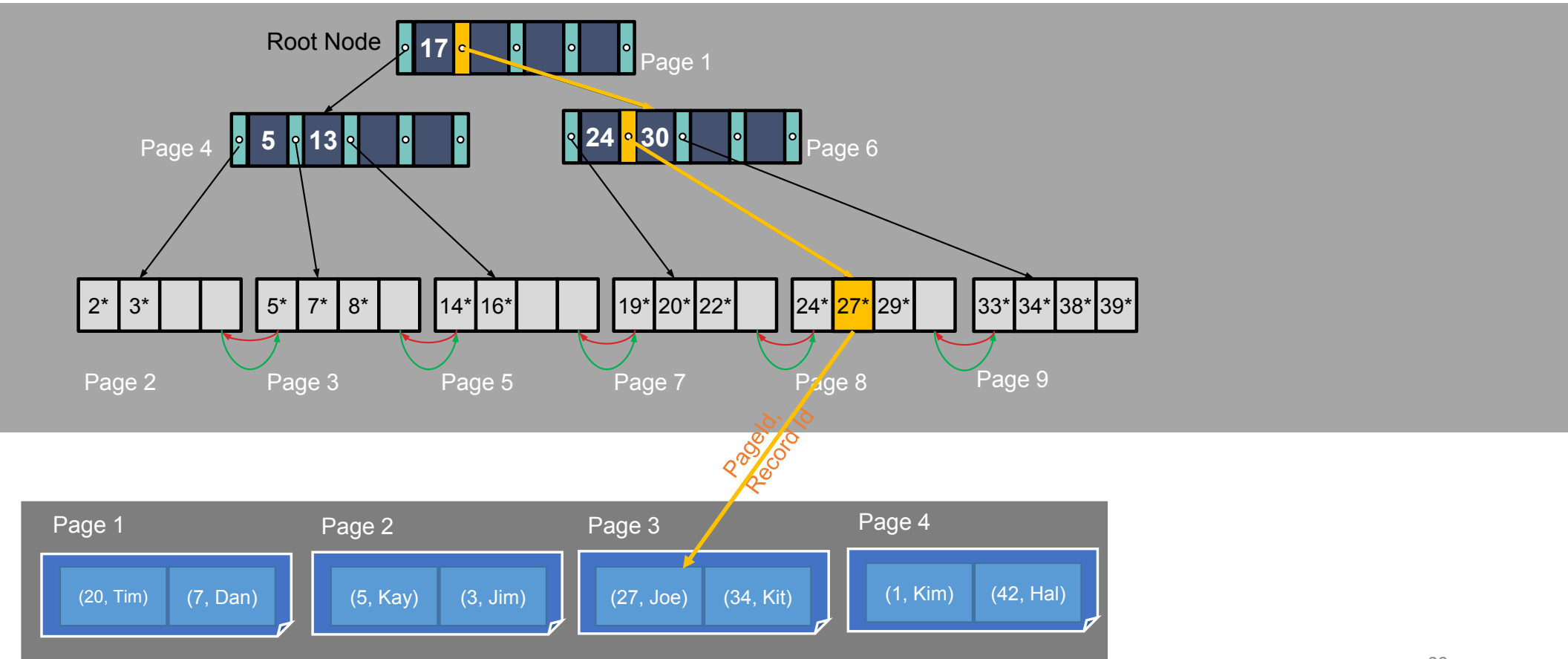
Data pages allocated dynamically

B+ Trees and Scale

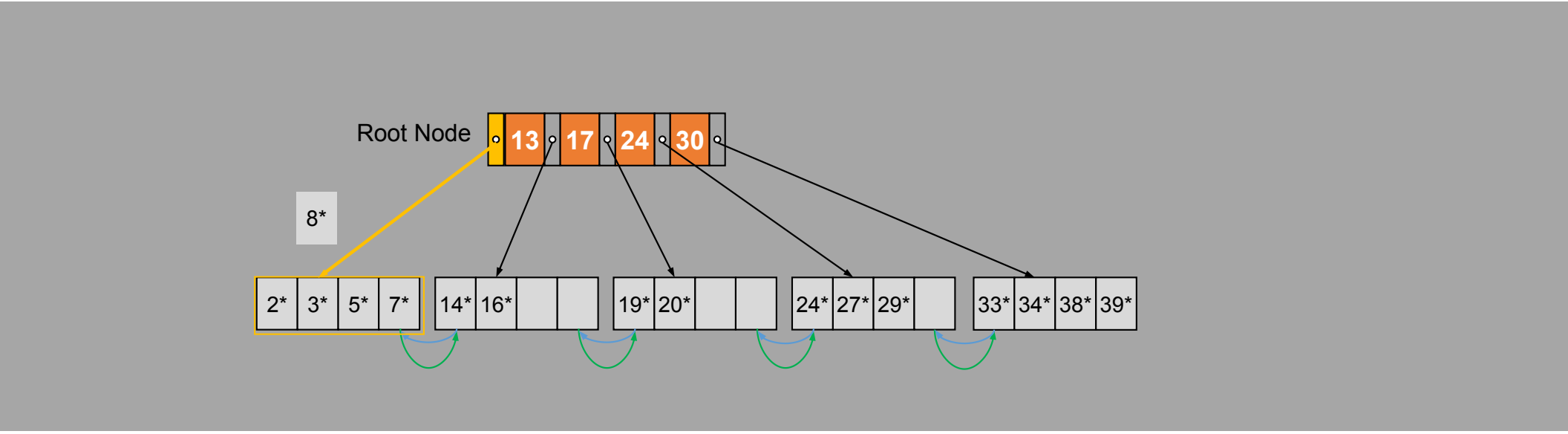


- How big is a height 3 B+ tree
  - d = 2 → Fan-out?
  - Fan-out = 2d + 1 = 5
  - **Height 3:** 5^3 x 4= 500 Records

Searching the B+ Tree: Fetch Data

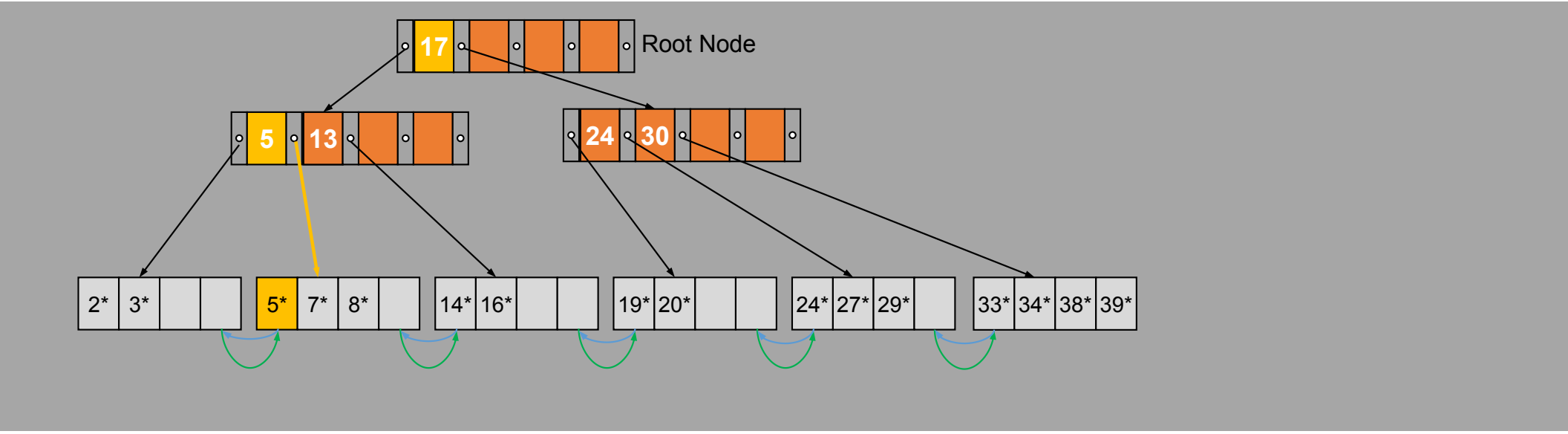


Inserting 8* into a B+ Tree: Insert



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

Copy up vs Push up!



- Notice:
  - The **leaf** entry (5) was **copied** up
  - The **index** entry (17) was **pushed** up

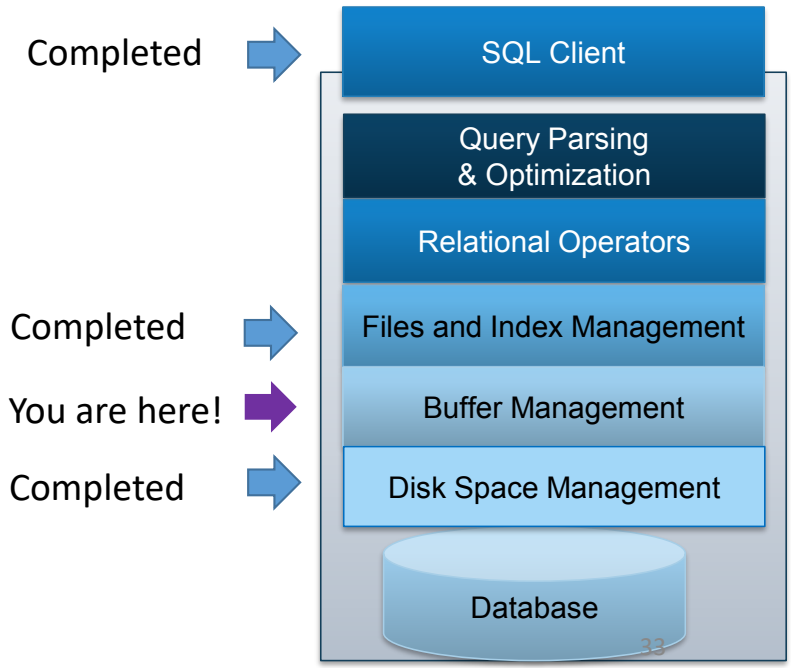
Check invariants

- **Key Invariant:**
  - Node[…, (K_L, P_L), …] ➔ K_L <= K for all K in P_L Sub-tree
- **Occupancy Invariant:**
  - d <= # entries <= 2d

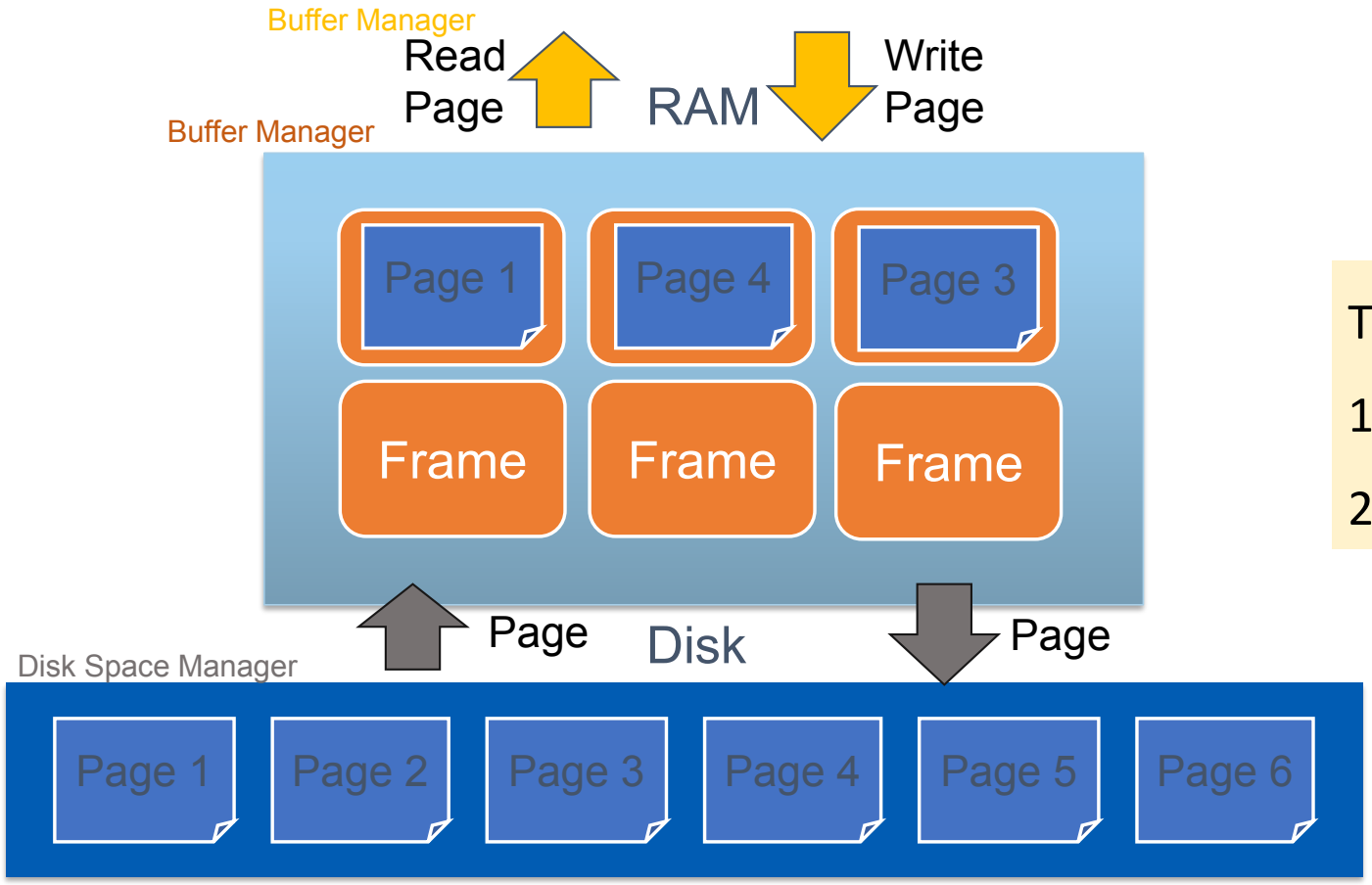
Three basic alternatives for data entries in any index

- Three basic alternatives for data entries in any index
  - Alternative 1: By Value
  - Alternative 2: By Reference
  - Alternative 3: By List of references
    - We'll look in the context of B+-trees, but applies to any index

4. Buffer Manager



APIs



Two questions:

1. Handling dirty pages
2. Page Replacement

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

LRU Replacement Policy

- Very common policy: intuitive and simple
  - Good for repeated accesses to popular pages (temporal locality)
  - Can be costly. Why?
    - Need to “find min” on the last used attribute (priority heap data structure)
- Approximate LRU: CLOCK policy

FrameId	PageId	Dirty?	Pin Count	Last Used
1	1	N	0	43
2	2	Y	1	21
3	3	N	0	22
4	6	N	2	11
5	4	N	0	24
6	5	N	0	15

Clock Policy State: Illustrated

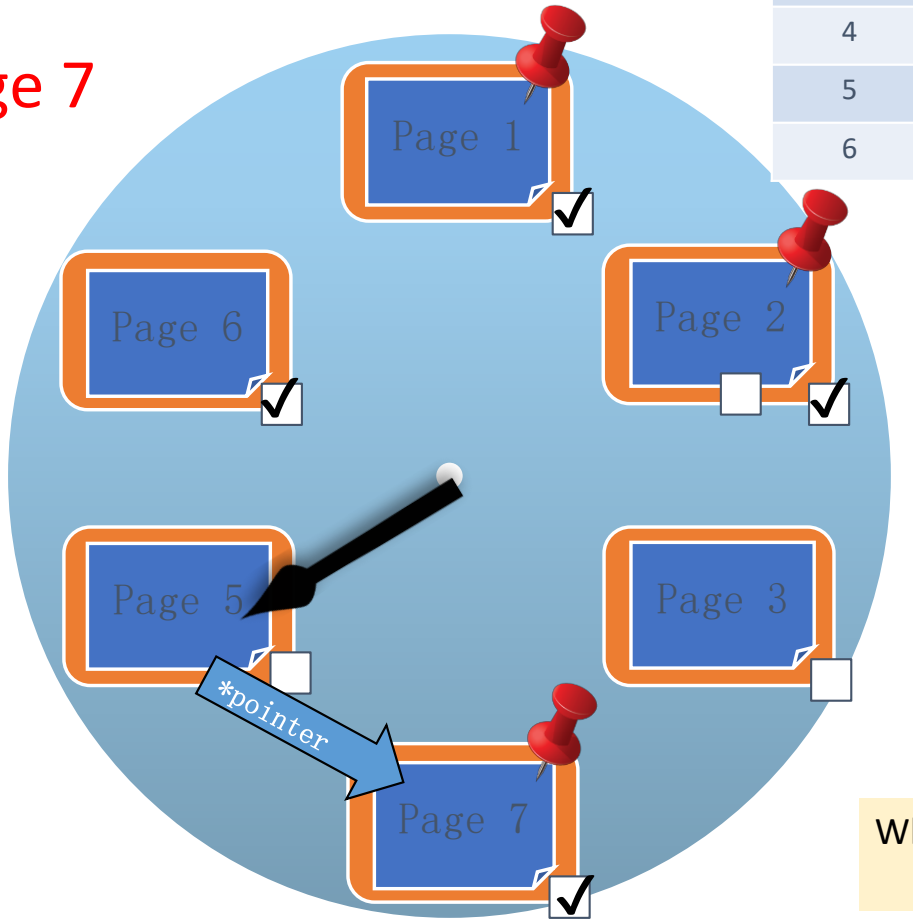
Request: Read page 7

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

Current frame not pinned
Ref bit unset:

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

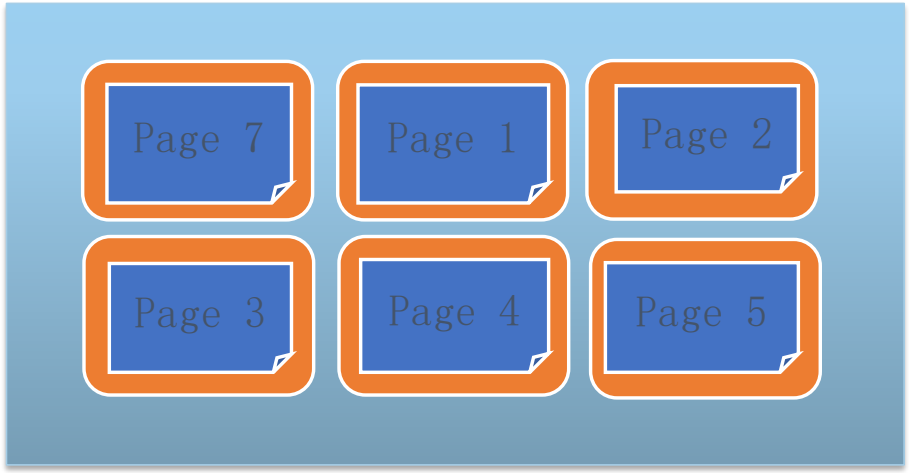


When might they perform poorly?

- repeated scans of big files

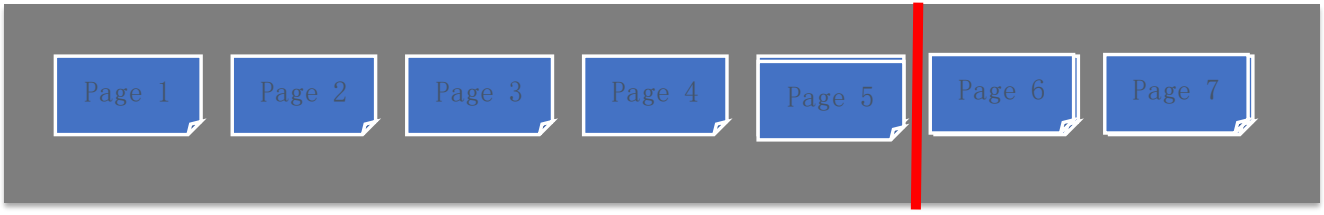
Repeated Scan (LRU): Read Page 5

- Cache Hits: 0
- Attempts: 12



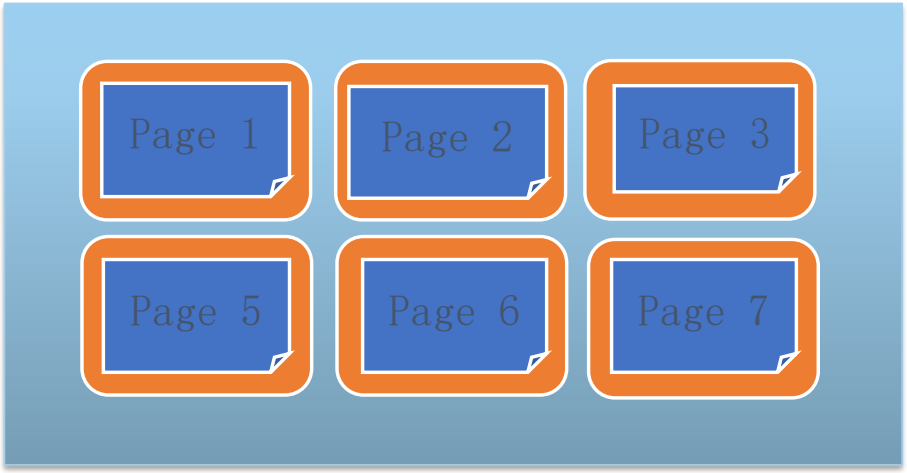
Get the picture? A worst-case scenario!
"Sequential Flooding"

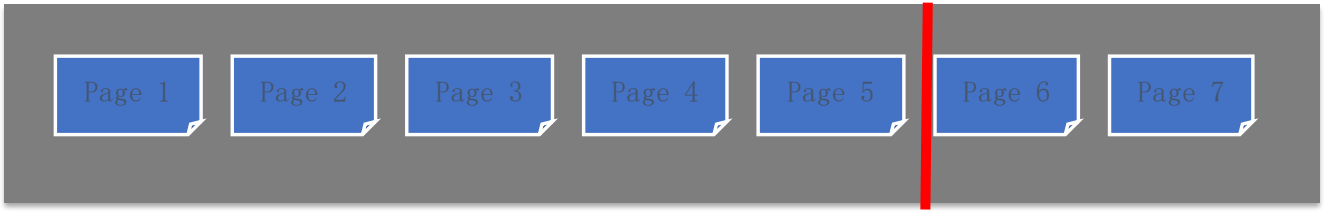
Disk Space Manager



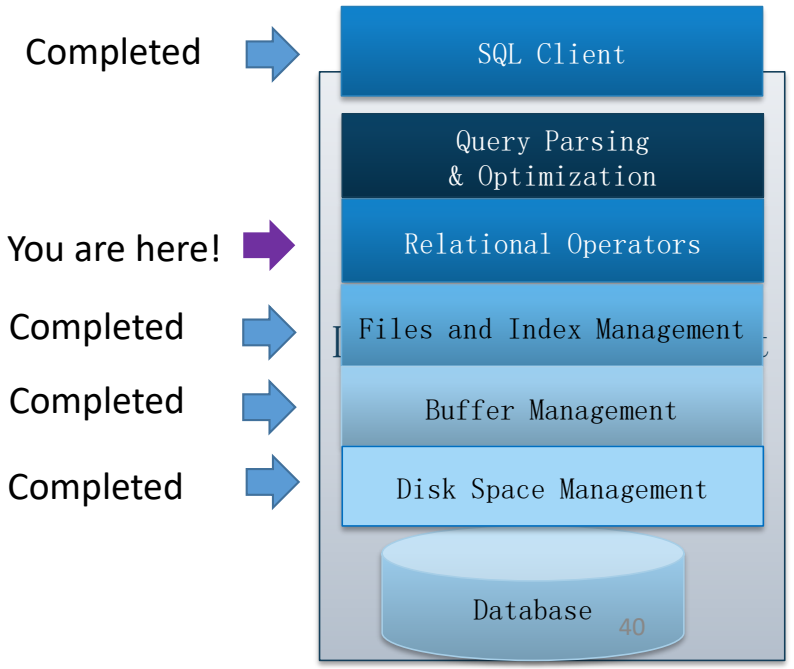
Repeated Scan (MRU): Read Page 5

- Cache Hits: 10
- Attempts: 19

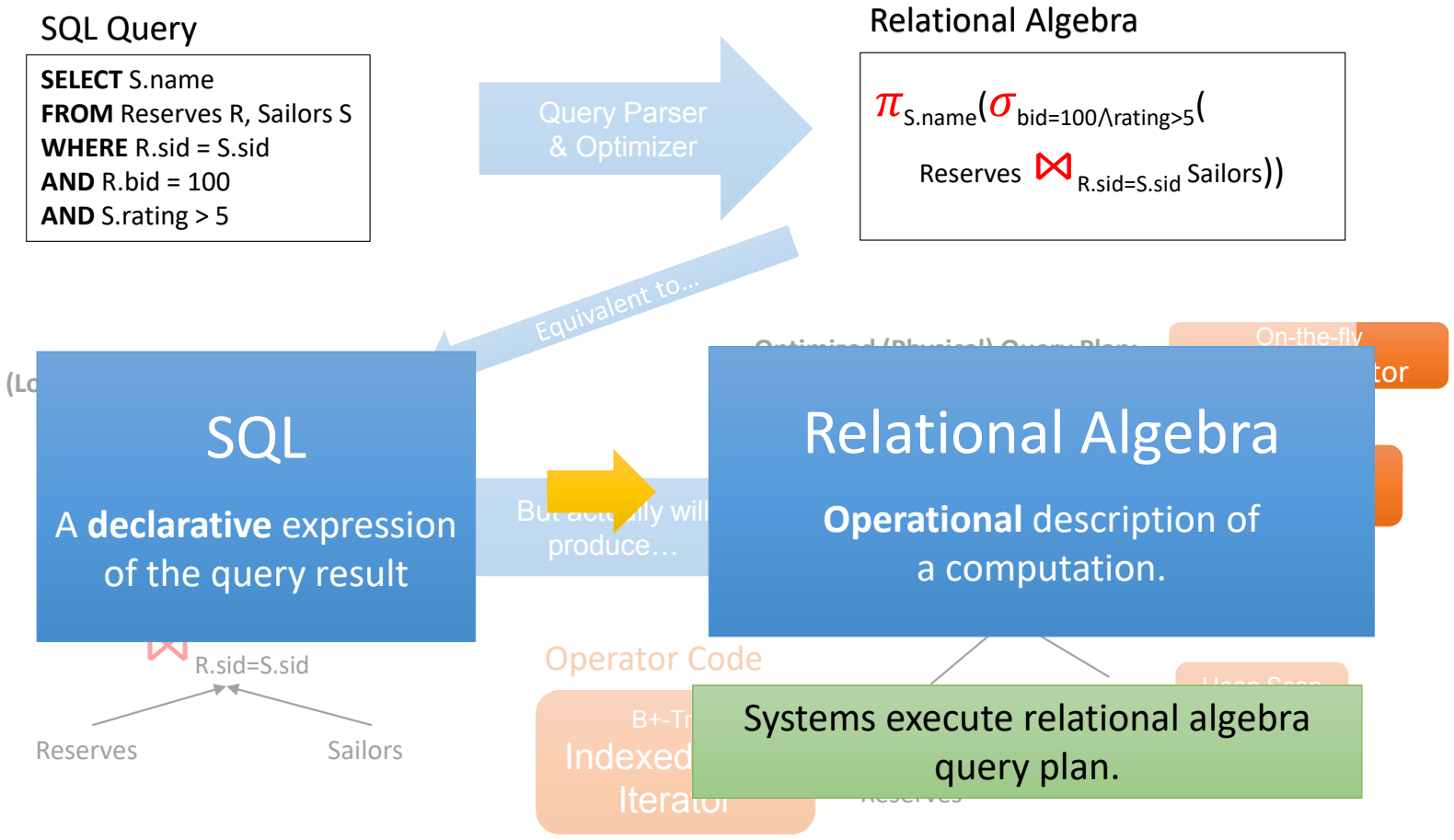




5. Relational Algebra



* SQL vs Relational Algebra



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

Sort ?

- "Rendezvous"
  - Eliminating duplicates (DISTINCT)
  - Grouping for summarization (GROUP BY)
  - Upcoming sort-merge join algorithm
- Ordering
  - Sometimes, output must be ordered (ORDER BY)
    - e.g., return results ranked in decreasing order of relevance
  - First step in bulk-loading tree indexes
- Problem: sort 100GB of data with 1GB of RAM.
  - why not virtual memory?

Hash?

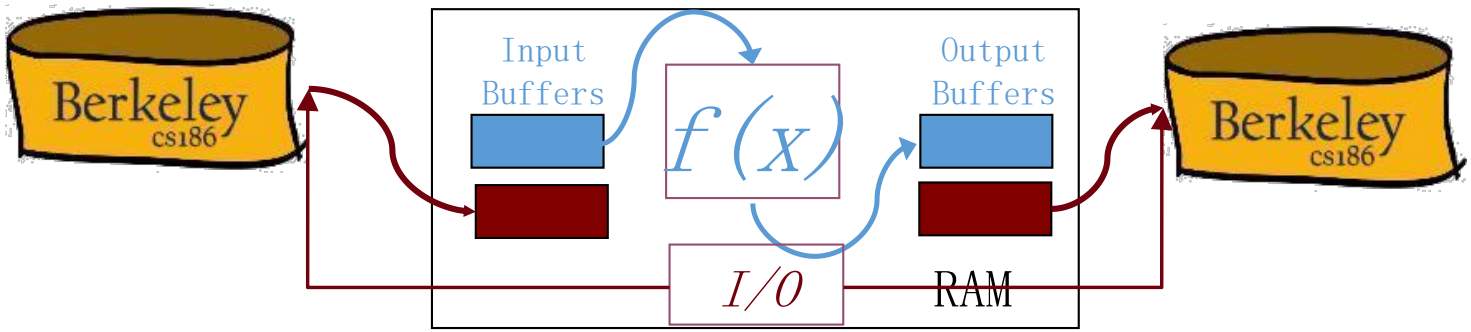
- Idea:
  - Many times we don't require order
  - E.g.: removing duplicates
  - E.g.: forming groups
- Often just need to rendezvous matches
- Hashing does this
  - But how to do it out-of-core??

Out-of-Core Algorithms

- Two themes
  1. Single-pass streaming data through RAM
  2. Divide (into RAM-sized chunks) and Conquer

Single-pass Streaming -- Double Buffering

- **Main thread** runs f(x) on one pair I/O bufs
- 2nd **I/O thread** drains/fills unused I/O bufs in parallel
  - Why is parallelism available?
  - Theme: I/O handling usually deserves its own thread
- Main thread ready for a new buf? Swap!

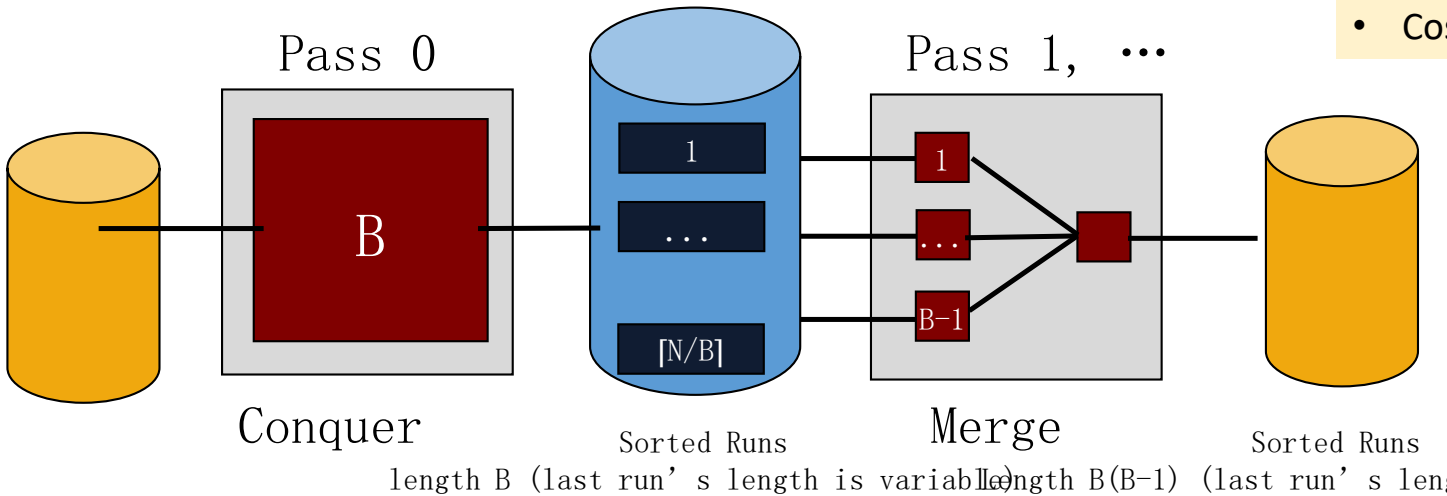


General External Merge Sort

- 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 N / B sorted runs of B pages each.
  - Pass 1, 2, …, etc.: merge B-1 runs at a time.

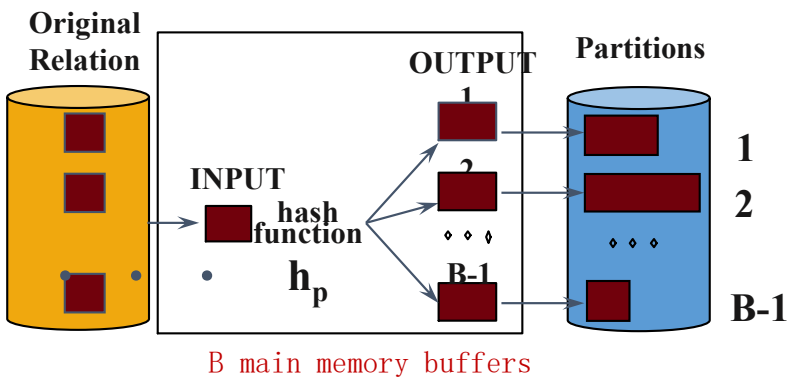
Cost
- Number of passes: $1 + \log_{B-1} N / B$
- Cost = 2N * (# of passes)



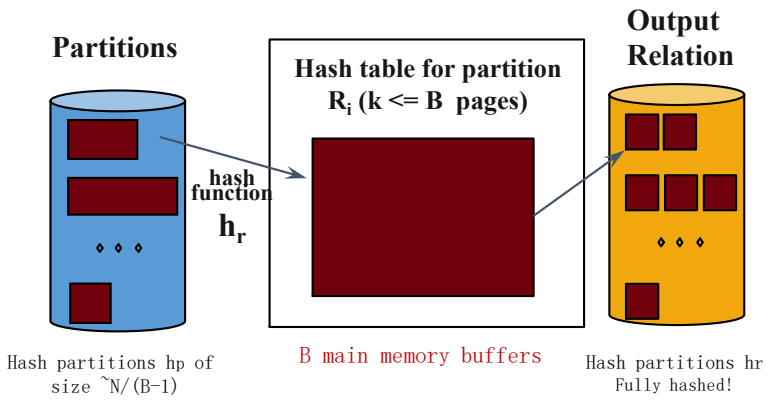
How big of a table can we sort in two passes?
- Answer: B(B-1).

Two Phases of Hash

- Partition:
  (Divide)



- Rehash:
  (Conquer)

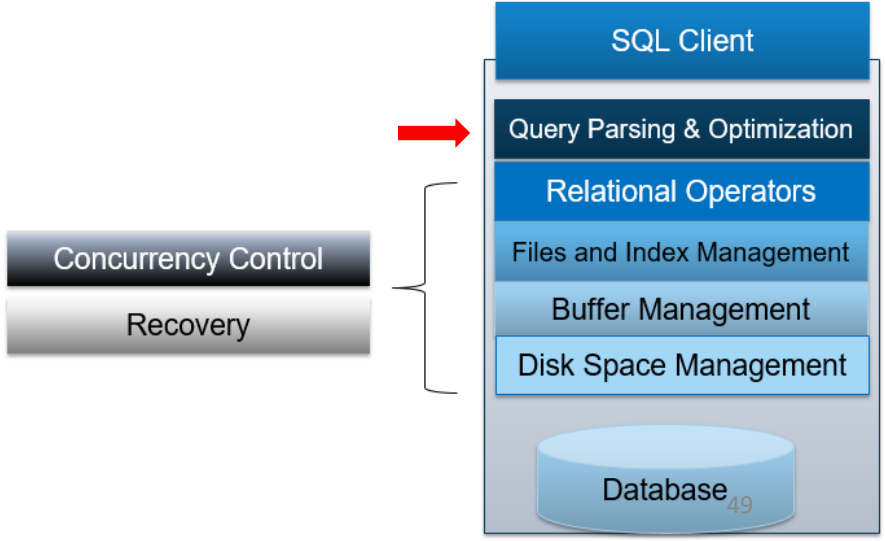


cost = 2*N*(#passes) = 4*N IO's
(includes initial read, final write)

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

- Answer: B(B-1).

7. Iterations and Joins

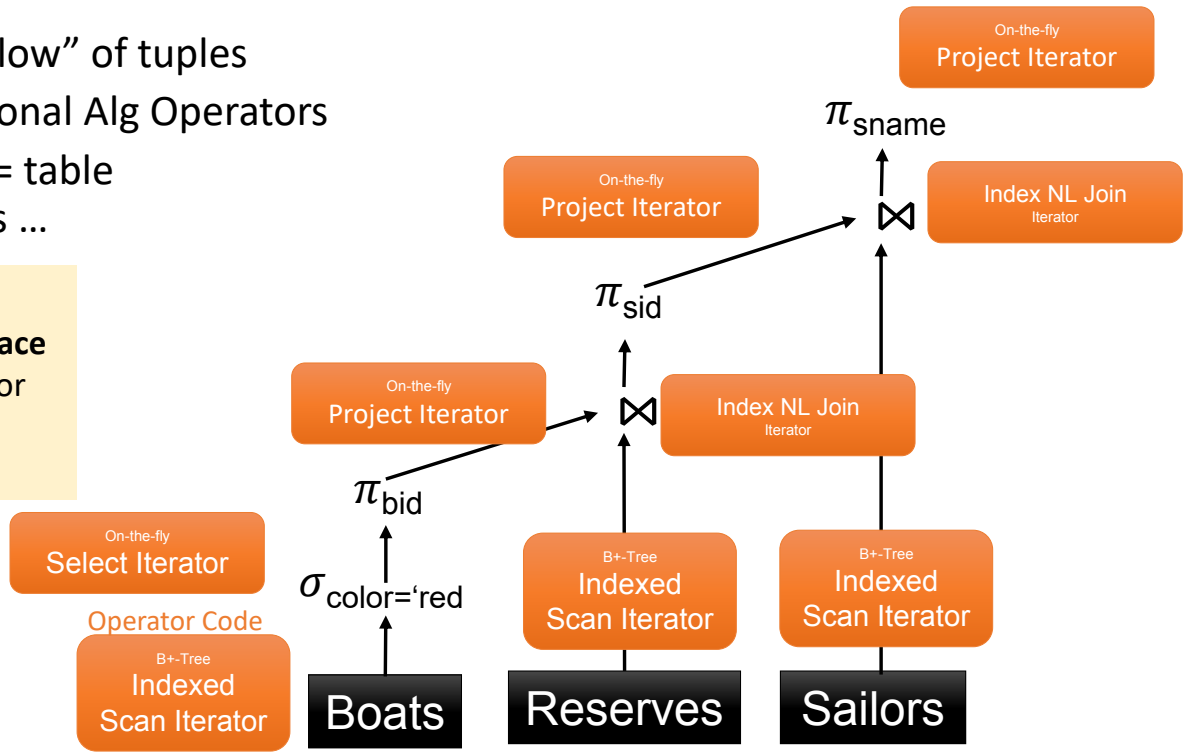


Query Executor Instantiates Operators

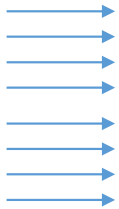
$$\pi_{\text{sname}}(\pi_{\text{sid}}(\pi_{\text{bid}}(\sigma_{\text{color='red'}}(\text{Boats})) \bowtie \text{Res}) \bowtie \text{Sailors})$$

- Query plan
  - Edges encode "flow" of tuples
  - Vertices = Relational Alg Operators
  - Source vertices = table access operators …

Each operator instance:
- Implements **iterator interface**
- Efficiently executes operator logic forwarding tuples to next operator



"Chunk" ~~"Block"~~ Nested Loop Join

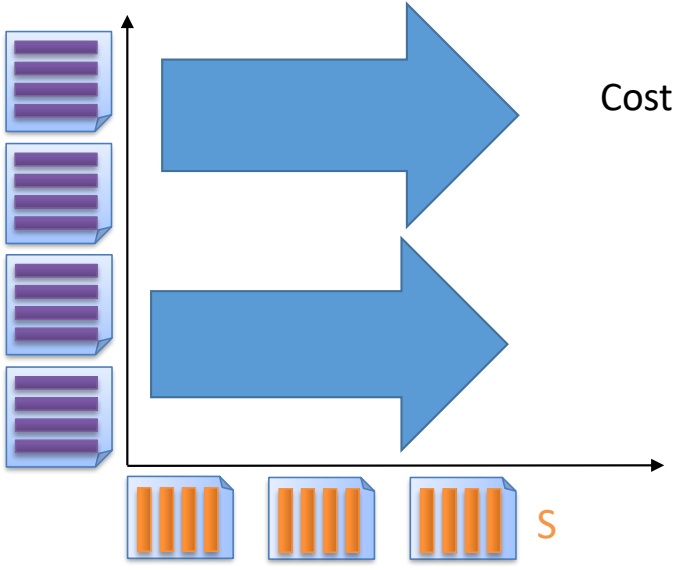


for each rchunk of B-2 pages of R:

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!)

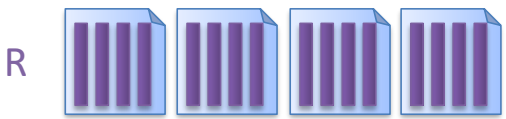
Which is the inner / outer relation?

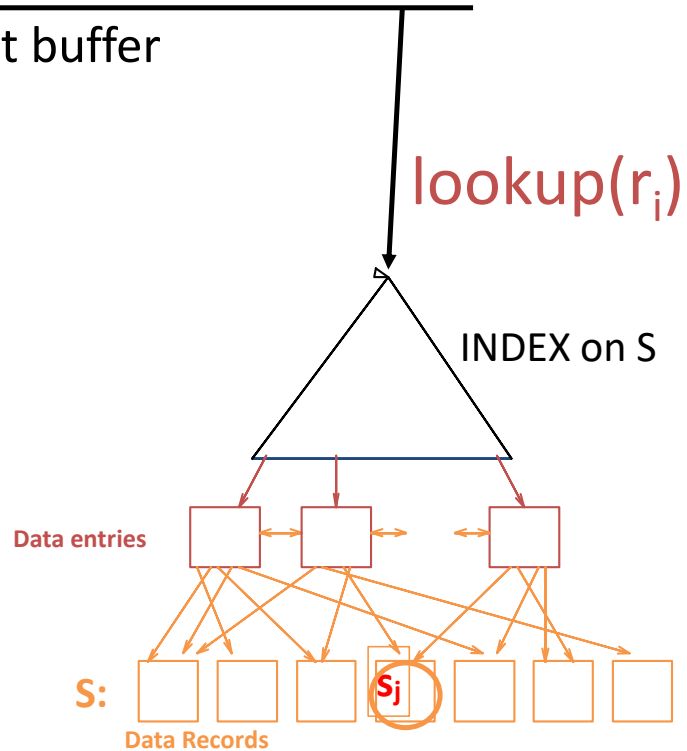
Index Nested Loops Join

foreach **tuple** r in R do

foreach **tuple** s in S where **ri == sj** do

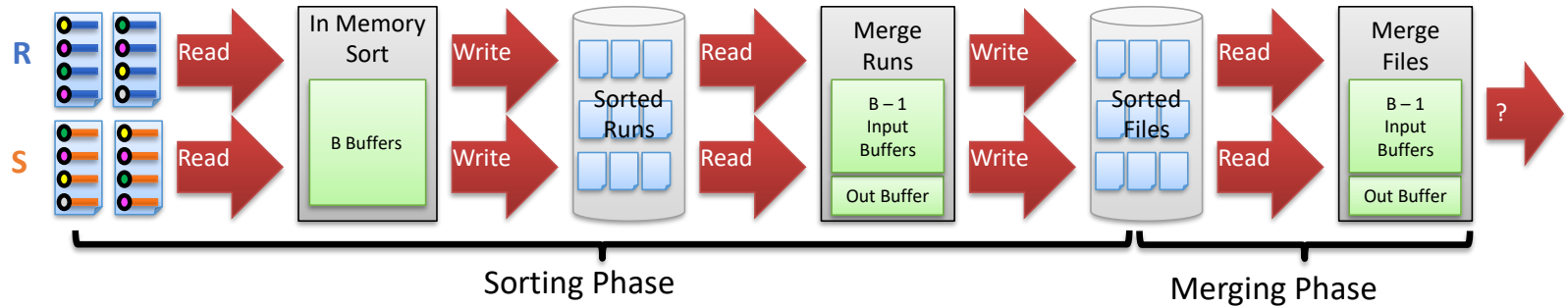
add <ri, sj> to result buffer





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

Cost of Sort-Merge Join

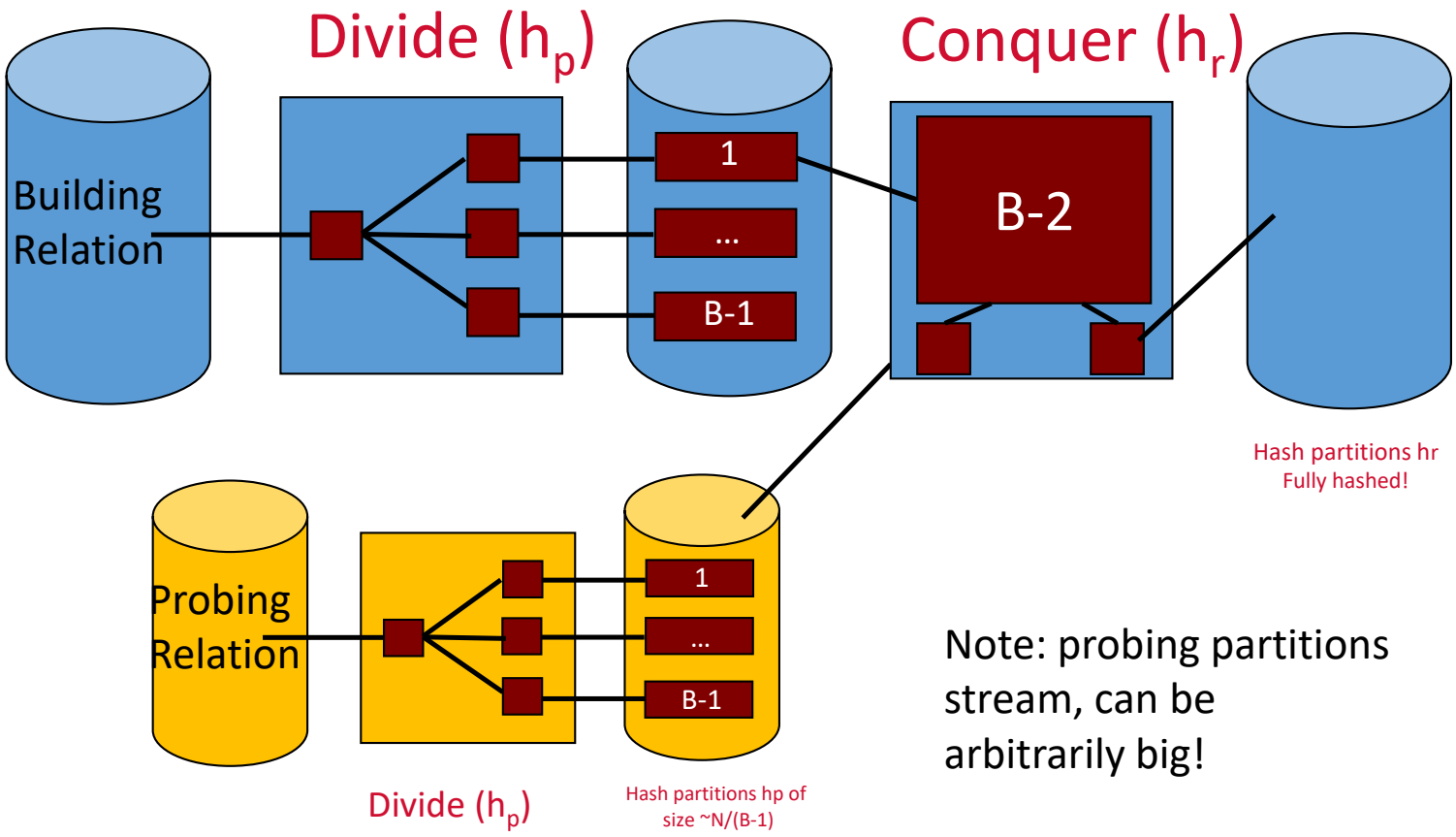


- Cost: Sort R + Sort S + ([R]+[S])
  - But in worst case, last term could be |R| *[S] (very unlikely!)
  - Q: what is worst case?

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

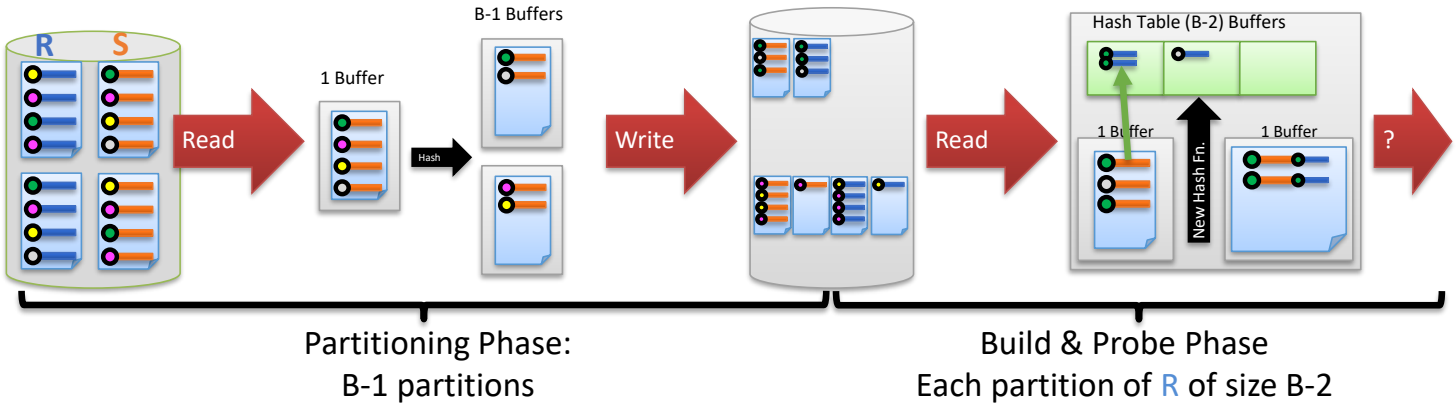
- Suppose buffer B > $\sqrt{(\max([R], [S]))}$
  - Both R and S can be sorted in 2 passes
  - 4*1000 + 4*500 + (1000 + 500) = 7500

Sketch of Grace Hash Join



Note: probing partitions stream, can be arbitrarily big!

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])

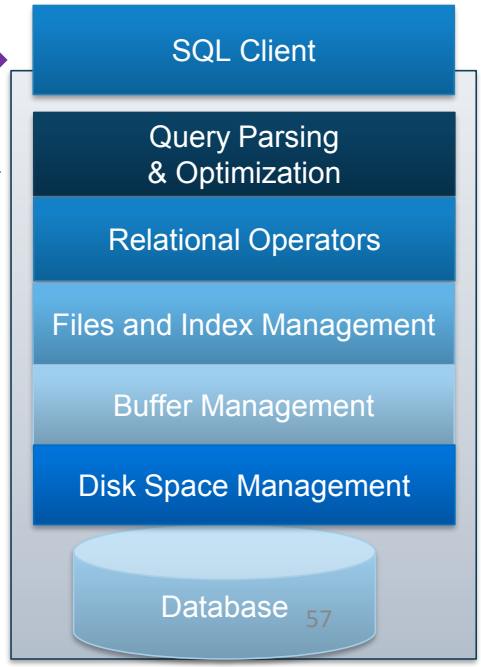
- **Memory Requirements?**
- Build hash table on R with uniform partitioning
  - **Partitioning Phase divides R into (B-1) runs of size [R] / (B-1)**
  - **Matching Phase requires each [R] / (B-1) < (B-2)**
  - **R < (B-1) (B-2) ≈ B^2**
- Note: no constraint on size of S (probing relation)!

Recap

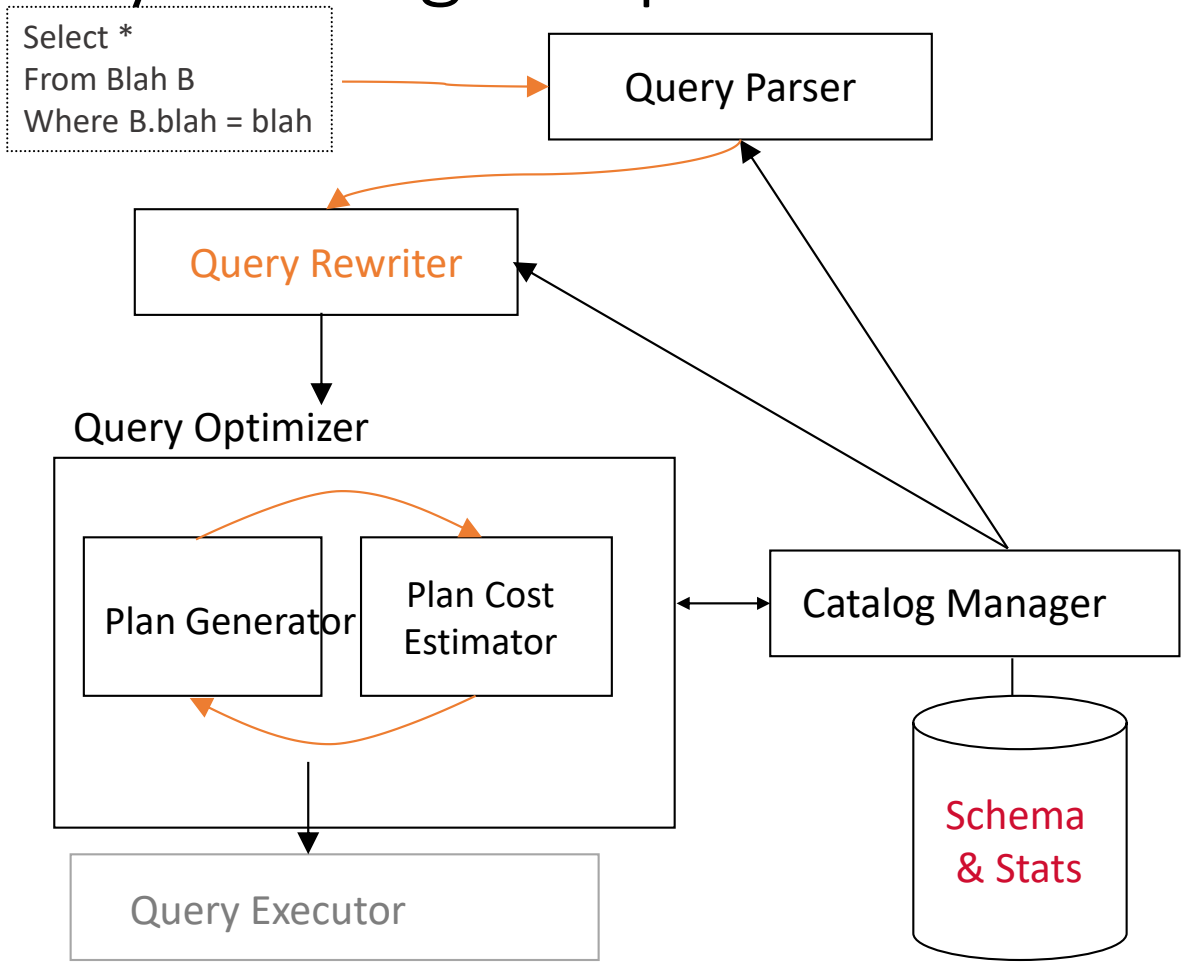
- Nested Loops Join
  - Works for arbitrary Θ
  - Make sure to utilize memory in blocks
- Index Nested Loops
  - For equi-joins
  - When you already have an index on one side
- Sort/Hash
  - For equi-joins
  - No index required
  - Hash better if one relation is much smaller than other
- No clear winners – may want to implement them all
- Be sure you know the cost model for each
  - You will need it for query optimization!

8. Query Optimization

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



Query Parsing & Optimization



We'll focus on **"System R" ("Selinger") optimizers**

Query Optimization: The Components

- Three beautifully orthogonal concerns:
  - Plan space:
    - for a given query, what plans are considered?
  - Cost estimation:
    - how is the cost of a plan estimated?
  - Search strategy:
    - how do we “search” in the “plan space”?

Plan Space Review

- For a SQL query, full plan space:
  - All equivalent relational algebra expressions
    - Based on the equivalence rules we learned
  - All mixes of physical implementations of those algebra expressions
- We might prune this space:
  - Selection/Projection pushdown
  - Left-deep trees only
  - Avoid cartesian products
- Along the way we may care about physical properties like sorting
  - Because downstream ops may depend on them
  - And enforcing them later may be expensive

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

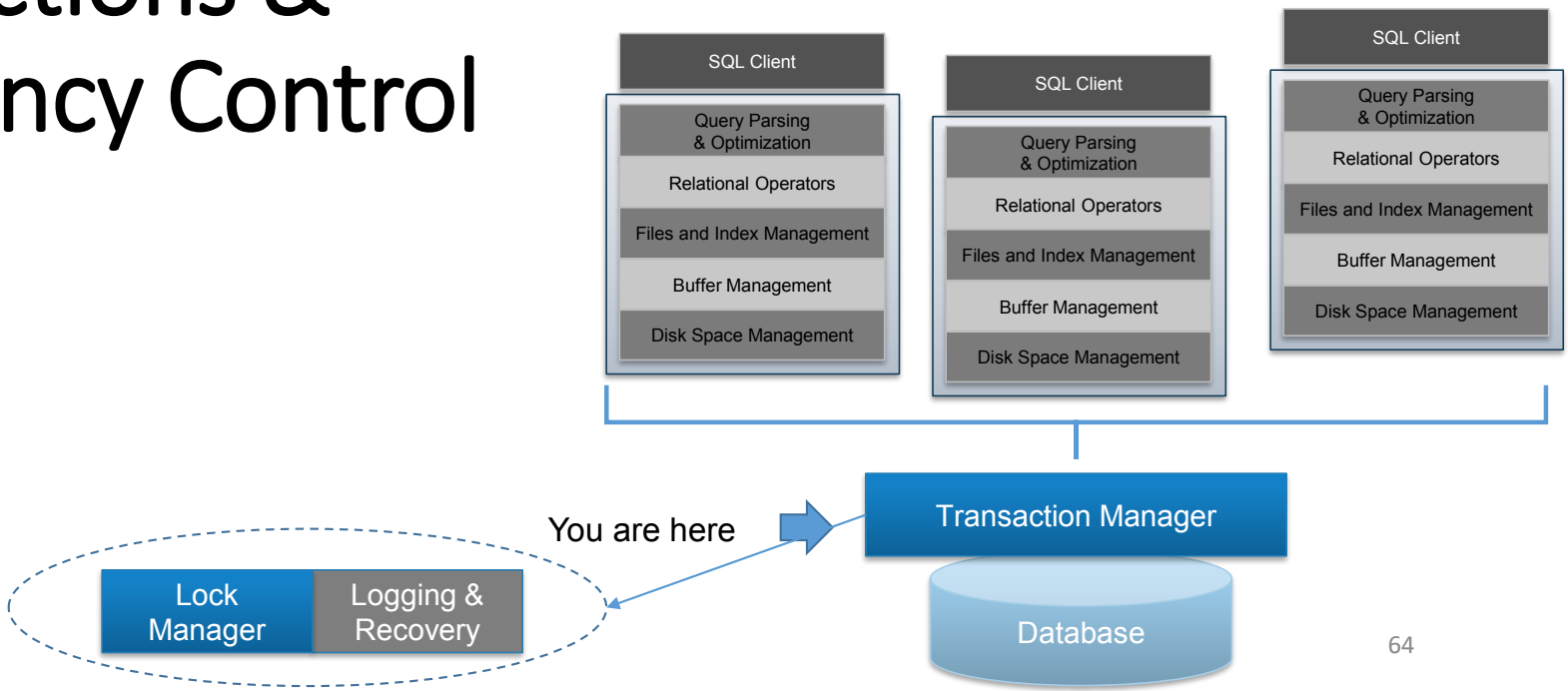
- Know how to compute selectivities for basic predicates
  - The original Selinger version
  - The histogram version

- **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

9. Transactions & Concurrency Control



ACID: High-Level Properties of Transactions

- **A tomicity:** *All* actions in the Xact happen, or *none* happen.
- **C onsistency:** If the DB *starts* out *consistent*, it *ends* up *consistent* at the end of the Xact
- **I solation:** Execution of *each* Xact is *isolated from* that of *others*
- **D urability:** If a Xact *commits*, its effects *persist*.

Note: This is a mnemonic, not a formalism. We'll do some formalisms shortly.

Serial Equivalence

- We need a “touchstone” concept for correct behavior
- **Definition**: **Serial schedule**
  - Each transaction runs from start to finish without any intervening actions from other transactions
- **Definition**: 2 schedules are **equivalent** if they:
  - involve the same transactions
  - each individual transaction’s actions are ordered the same
  - both schedules leave the DB in the same final state
- **Definition**: Schedule S is **serializable** if:
  - S is equivalent to some serial schedule

Conflict Serializable Schedules

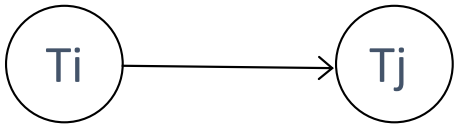
- **Definition: Two schedules are *conflict equivalent* if:**
  - They involve the same actions of the same transactions, and
  - Every pair of conflicting actions is ordered the same way

- **Definition: Schedule S is *conflict serializable* if:**
  - S is conflict equivalent to some serial schedule
  - Implies S is also Serializable

Note: some serializable schedules are NOT conflict serializable
  - Conflict serializability gives false negatives as a test for serializability!
  - The cost of a conservative test
  - A price we pay to achieve efficient enforcement

Definition: Two operations conflict if they:
Are by different transactions,
Are on the same object,
At least one of them is a write.

Conflict Dependency Graph



- **Dependency Graph:**
  - One node per Xact
  - Edge from Ti to Tj if:
    - An operation Oi of Ti conflicts with an operation Oj of Tj and
    - Oi appears earlier in the schedule than Oj

- **Theorem: Schedule is conflict serializable if and only if its dependency graph is acyclic.**

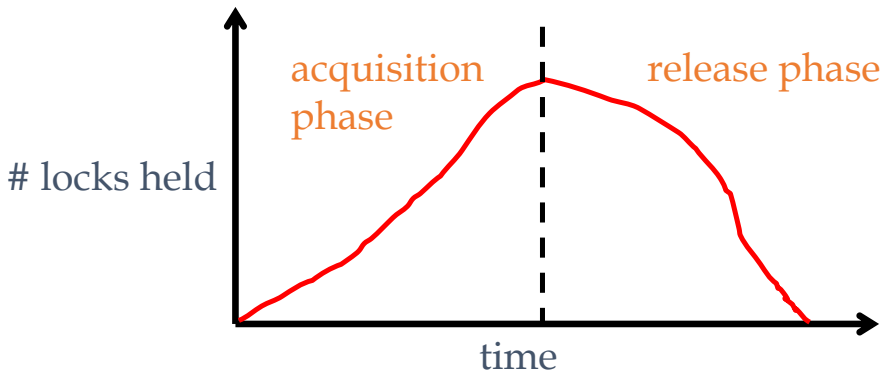
  Proof Sketch: Conflicting operations prevent us from "swapping" operations into a serial schedule

Two Phase Locking (2PL), Part 2

- Rules:
  - Xact must obtain a S (shared) lock before reading, and an X (exclusive) lock before writing.
  - **Xact cannot get new locks after releasing any locks**

Lock Compatibility Matrix

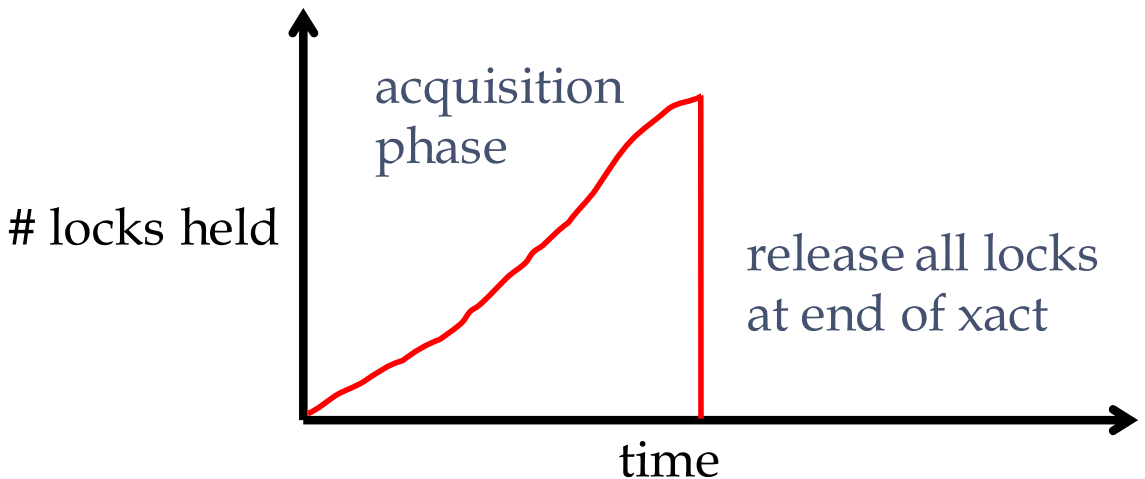
	S	X
S	√	–
X	–	–



- **2PL guarantees conflict serializability**
- But, does not prevent **cascading aborts**

Strict Two Phase Locking

- Same as 2PL, except all locks released together when transaction completes
  - (i.e.) either
    - Transaction has committed (all writes durable), OR
    - Transaction has aborted (all writes have been undone)



Lock Management

- Lock and unlock requests handled by Lock Manager
- LM maintains a hashtable, keyed on names of objects being locked.
- LM keeps an entry for each currently held lock
- Entry contains
  - Granted set: Set of xacts currently granted access to the lock
  - Lock mode: Type of lock held (shared or exclusive)
  - Wait Queue: Queue of lock requests

	Granted Set	Mode	Wait Queue
A	{T1, T2}	S	T3(X) ← T4(X)
B	{T6}	X	T5(X) ← T7(S)

When lock request arrives:

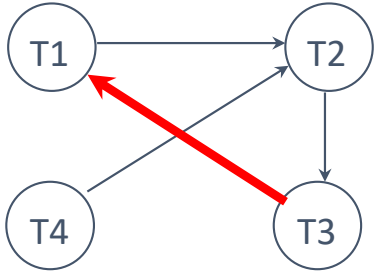
Does any xact in Granted Set or Wait Queue want a conflicting lock?

If no, put the requester into "granted set" and let them proceed

If yes, put requester into wait queue (typically FIFO)

Deadlocks, cont

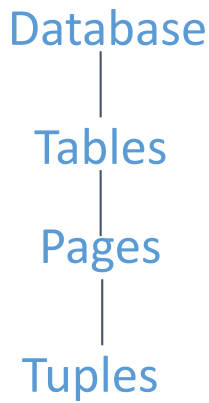
- **Deadlock: Cycle of Xacts waiting for locks to be released by each other.**
- **Three ways of dealing with deadlocks:**
  - Prevention
  - Avoidance
  - Detection and Resolution
- **Many systems just punt and use timeouts**
  - What are the dangers with this approach?



Deadlock Detection
- Create and maintain a **"waits-for" graph**
- Periodically check for cycles in a graph

Multiple Granularity Locking Protocol

- Each Xact starts from the root of the hierarchy.
- To get S or IS lock on a node, must hold IS or IX on parent node.
  - What if Xact holds S on parent? SIX on parent?
- To get X or IX or SIX on a node, must hold IX or SIX on parent node.
- Must release locks in bottom-up order.

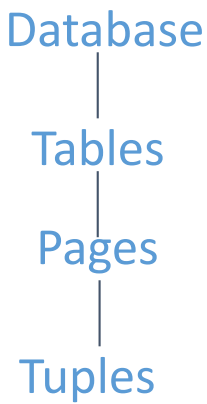


- 2-phase and lock compatibility matrix rules enforced as well
- Protocol is correct in that it is *equivalent to directly setting locks at leaf levels of the hierarchy*.

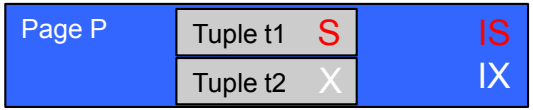
Lock Compatibility Matrix

- IS – Intent to get S lock(s) at finer granularity.
- IX – Intent to get X lock(s) at finer granularity.
- SIX mode: Like S & IX at the same time.

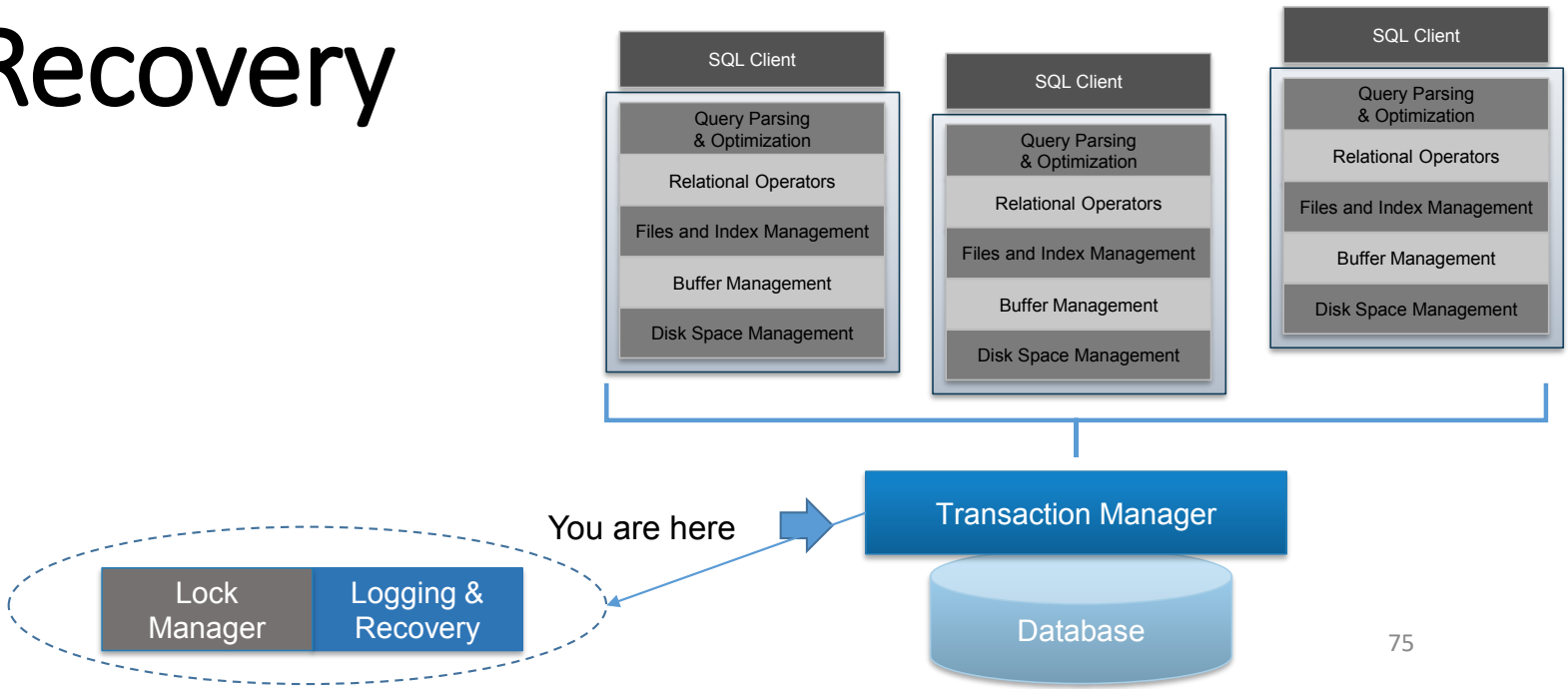
	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false



Handy simple case to remember:
Could 2 intent locks be compatible?



10. Recovery


SQL Client
Query Parsing & Optimization
Relational Operators
Files and Index Management
Buffer Management
Disk Space Management
SQL Client
Query Parsing & Optimization
Relational Operators
Files and Index Management
Buffer Management
Disk Space Management
SQL Client
Query Parsing & Optimization
Relational Operators
Files and Index Management
Buffer Management
Disk Space Management
You are here
Transaction Manager
Lock Manager
Logging & Recovery
Database

Preferred Policy: Steal/No-Force

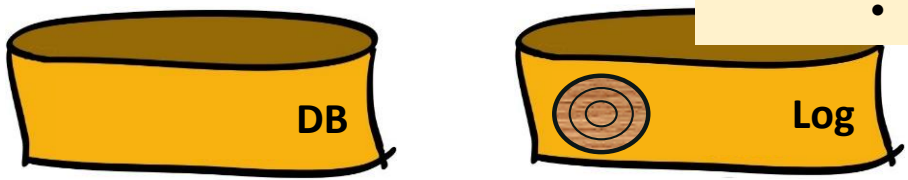
- Most complicated, but highest performance.
- **NO FORCE** (complicates enforcing Durability)
  - Problem: System crash before dirty buffer page of a committed transaction is flushed to DB disk.
  - Solution: Flush as little as possible, in a convenient place, prior to commit. Allows REDOing modifications.
- **STEAL** (complicates enforcing Atomicity)
  - What if a Xact that flushed updates to DB disk aborts?
  - What if system crashes before Xact is finished?
  - Must remember the old value of flushed pages
    - (to support UNDOing the write to those pages).

This is a dense slide … and the crux of the lecture.
Read it over carefully, and return to it later!

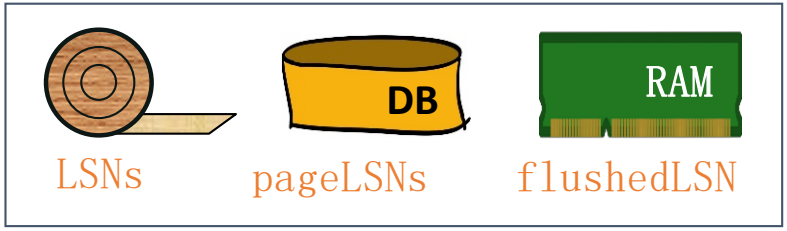
Write-Ahead Logging (WAL)

- The **Write-Ahead Logging Protocol**:
  1. Must **force** the **log record** for an update **before** the corresponding **data page** gets to the DB disk.
  2. Must **force all log records** for a Xact **before commit**.
     - I.e. transaction is not committed until all of its log records including its "commit" record are on the stable log.
- #1 (with **UNDO** info) helps guarantee Atomicity.
- #2 (with **REDO** info) helps guarantee Durability.
- This allows us to implement Steal/No-Force

Log: An **ordered list** of log records to allow REDO/UNDO
- Log record contains:
  - **<XID, pageID, offset, length, old data, new data>**



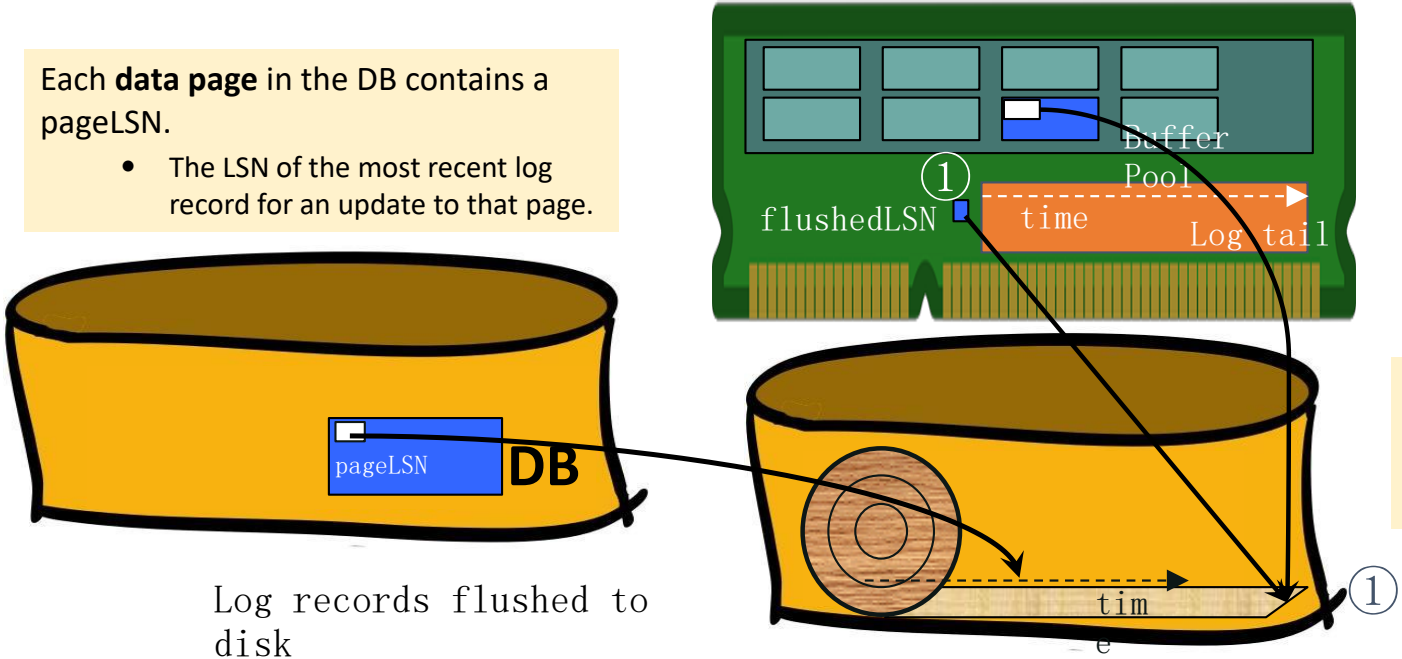
WAL & the Log



- Log: an ordered file, with a write buffer ("tail") in RAM.
- Each log record has a **Log Sequence Number** (LSN).
  - LSNs unique and increasing.
  - **flushedLSN** tracked in RAM

Each **data page** in the DB contains a pageLSN.

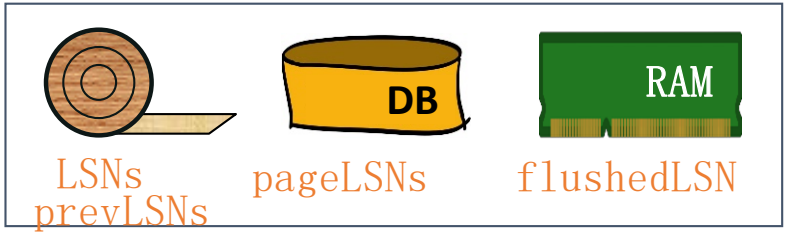
- The LSN of the most recent log record for an update to that page.



WAL: Before page i is flushed to DB, log must satisfy:

$$\textbf{pageLSN}_i \leqslant \textbf{flushedLSN}$$

ARIES Log Records



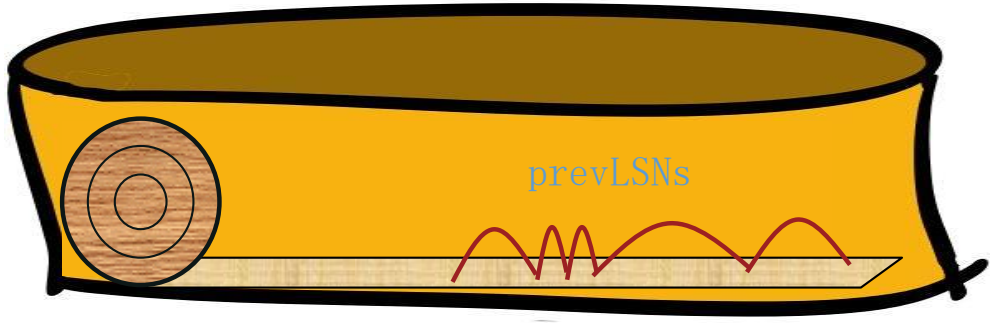
- Exactly how is logging (and recovery!) done?
  - The ARIES algorithm from IBM.

- **prevLSN** is the LSN of the previous log record written by this **XID**
  - So records of an Xact form a linked list backwards in time

LogRecord fields:
- LSN
- prevLSN
- XID
- typ
- pageID

update records only:
- length
- offset
- before-image
- after-image

Possible log record types:
- Update, Commit, Abort
- Checkpoint (for log maintainence)
- Compensation Log Records (CLRs)
  - (for UNDO actions)
- End (end of commit or abort)



Other Log-Related State

- Two in-memory tables:
- Transaction Table
  - One entry per currently active Xact.
    - removed when Xact commits or aborts
  - Contains:
    - **XID**
    - **Status** (running, committing, aborting)
    - **lastLSN** (most recent LSN written by Xact).
- Dirty Page Table
  - One entry per dirty page currently in buffer pool.
  - Contains **recLSN**
    - LSN of the log record which first caused the page to be dirty.

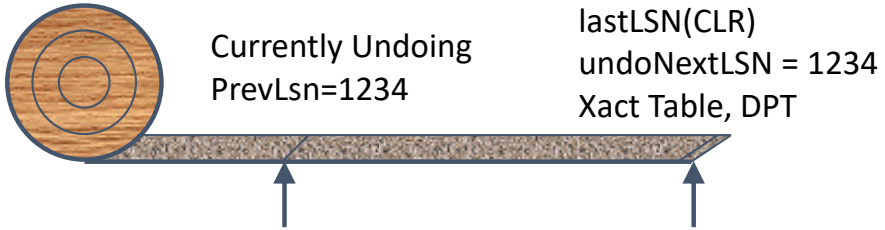
Transaction Table

XID	Status	lastLSN
1	R	33
2	C	42

Dirty Page Table

PageID	recLSN
46	11
63	24

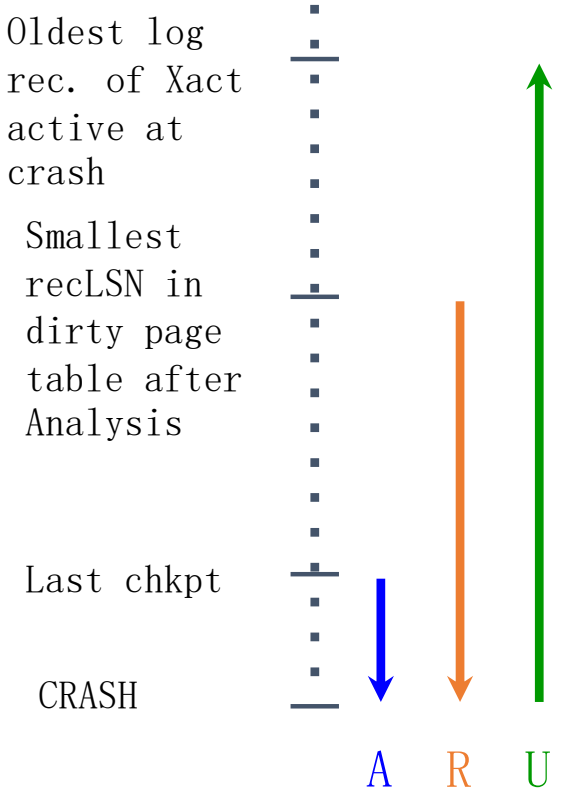
Checkpointing



- Conceptually, keep log around for all time.
  - Performance/implementation problems…
- Periodically, the DBMS creates a **checkpoint**
  - Minimizes recovery time after crash. Write to log:
    - **begin_checkpoint** record: Indicates when chkpt began.
    - **end_checkpoint** record: Contains current Xact table DPT
    - . A "**fuzzy checkpoint**": Other Xacts continue to run;
      - So all we know is that these tables are after the time of the begin_checkpoint record.
    - Store LSN of most recent chkpt record in a safe place
    - (**master record**, often block 0 of the log file).

Crash Recovery: Big Picture

- Start from a **checkpoint**
  - found via master record.
- Three phases. Need to do:
  - **Analysis** - Figure out which Xacts committed since checkpoint, which failed.
  - **REDO** all actions.
    - (repeat history)
  - **UNDO** effects of failed Xacts.



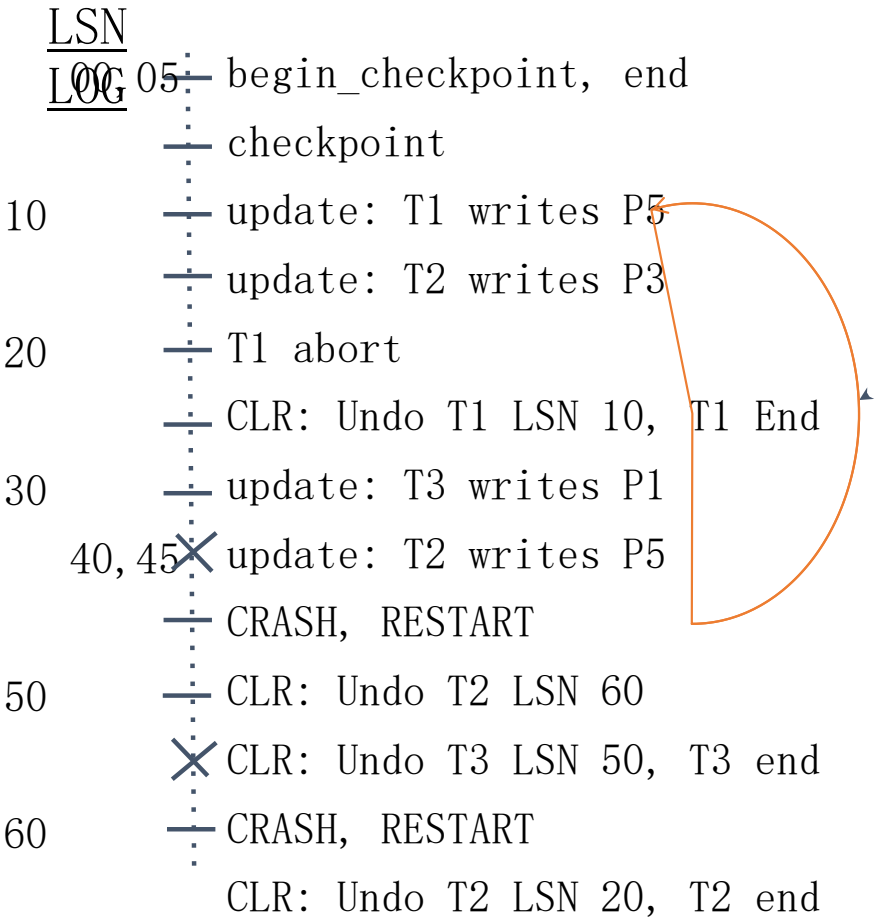
Example: Crash During Restart!

Xact Table
lastLSN
status

Dirty Page Table
recLSN

flushedLSN

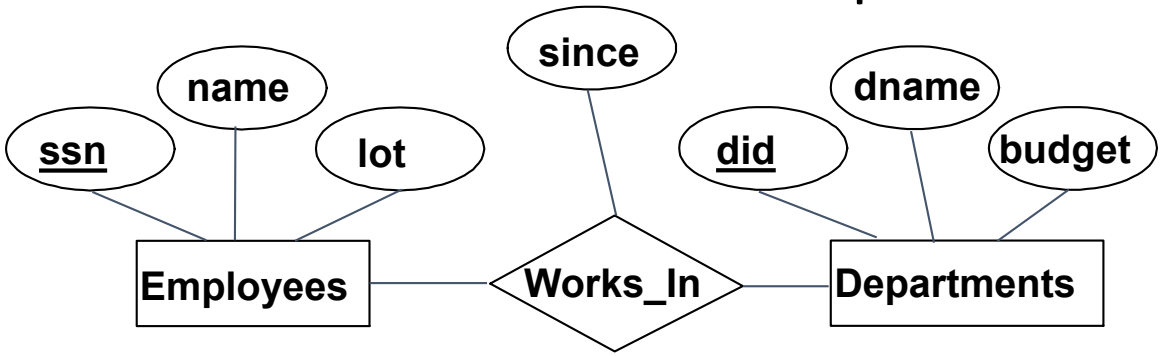
ToUndo



Using pencil and paper, run the ARIES recovery algorithm on this log, assuming you have access to a master record pointing to LSN 05. Maintain all the state on the left as you go!

11. ER Modeling & FD

ER Model Basics: Relationships



Entity:

- A real-world object described by a set of attribute values.

Entity Set: A collection of similar entities.

- Each entity set has a key (underlined)

Relationship: Association among two or more entities.

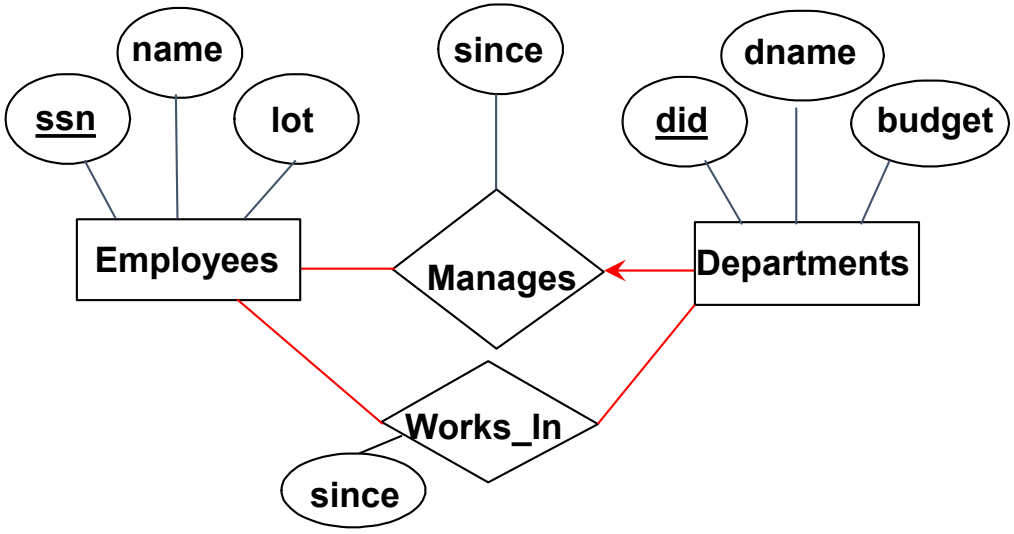
- Relationships can have their own attributes.

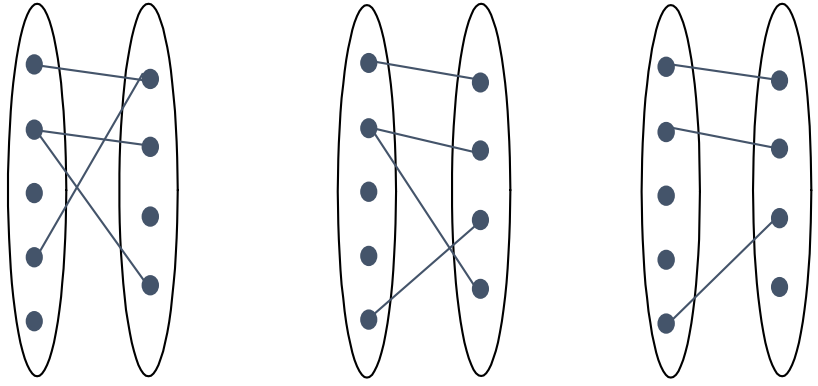
Relationship Set: Collection of similar relationships.

- An n-ary relationship set R relates n entity sets E1 ... En

Key Constraints

- An employee can work in **many** departments; a dept can have **many** employees.
- In contrast, each dept has **at most one** manager, according to the *key constraint* on **Department** in the **Manages** relationship set.
- A **key constraint** gives a 1-to-many relationship.





Many-to-Many

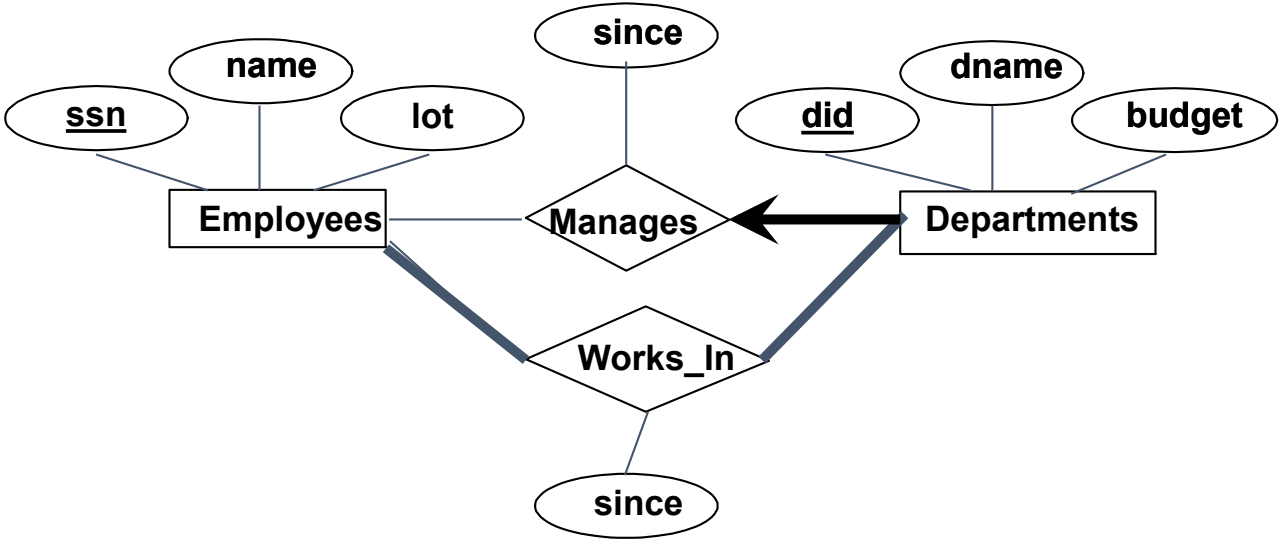
1-to-Many

Many-to-1

1-to-1

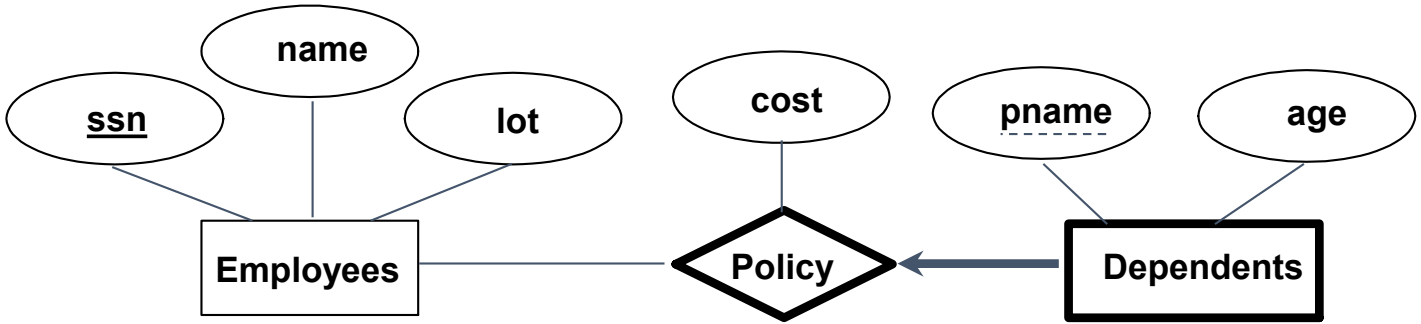
Participation Constraints

- Does every employee work in a department?
- If so: a **participation constraint**
  - participation of Employees in Works_In is total (vs. partial)
  - What if every department has an employee working in it?
- Basically means **at least one.**



Weak Entities

- A **weak entity** can be identified uniquely only by considering the primary key of another (owner) entity.
  - Owner entity set and weak entity set must participate in a one-to-many relationship set (one owner, many weak entities).
  - Weak entity set must have total participation in this identifying relationship set.



- Weak entities have only a “partial key” (dashed underline)

Important: key terminology

- Question: How are FDs related to primary keys?
  - **Primary Keys are special cases of FDs**
  - K → {all attributes}
- Superkey: a set of columns that determines all the columns in its table
  - K → {all attributes}. (Also sometimes just called a key)
- Candidate Key: a **minimal** set of columns that determines all columns in its table
  - K → {all attributes}
  - For any L ⊂ K, L !→ {all attributes} (minimal)
- Primary Key: a single chosen candidate key
- Index/sort "key" : columns used in an index or sort.
  - Unrelated to FDs, dependencies.

Rules of Inference

- **Armstrong's Axioms** (X, Y, Z are sets of attributes):
  - *Reflexivity*: If $X \supseteq Y$, then $X \rightarrow Y$
  - *Augmentation*: If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z
  - *Transitivity*: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

- Sound and complete inference rules for FDs!
  - using AA you get only the FDs in F+ and all these FDs.

- Some additional rules (that follow from AA):
  - *Union*: If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
  - *Decomposition*: If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$
  - See if you can prove these!

Attribute Closure

- Computing closure F+ of a set of FDs F is hard:
  - exponential in # attrs!

- Typically, just check if X → Y is in F+. Efficient!
  - Compute attribute closure of X (denoted X+) wrt F.
    X+ = Set of all attributes A such that X → A is in F+
    - X+ := X
    - Repeat until no change (fixpoint):
      for U → V ⊆ F,
      if U ⊆ X+, then add V to X+
  - Check if Y is in X+
  - Approach can also be used to check for keys of a relation.
    - If X+ = R, then X is a superkey for R.
    - Q: How to check if X is a "candidate key" (minimal)?
    - A: For each attribute A in X, check if (X-A)+ = R

Decomposition into BCNF

- Consider relation R with FDs F.
- If X → Y violates BCNF, decompose R into R - Y and XY (guaranteed to be loss-less).
  - Repeated application of this idea will give us a collection of relations that are in BCNF
  - Lossless join decomposition, and guaranteed to terminate.

- e.g., CSJDPQV, key C, JP → C, SD → P, J → S
  - {contractid, supplierid, projectid,deptid,partid, qty, value}
  - To deal with SD → P, decompose into SDP, CSJDQV.
  - To deal with J → S, decompose CSJDQV into JS and CJDQV
  - So we end up with: SDP, JS, and CJDQV

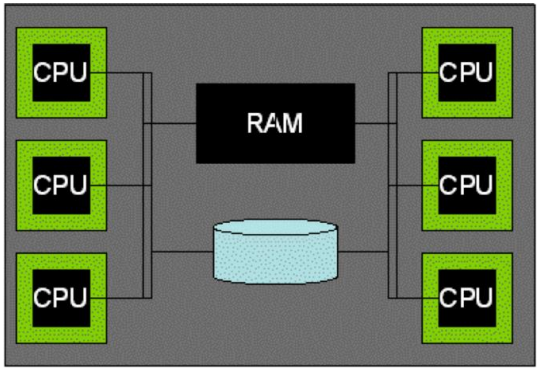
- Note: several dependencies may cause violation of BCNF.
- The order in which we "deal with" them could lead to very different sets of relations!

R is in BCNF if the only non-trivial FDs over R are key constraints

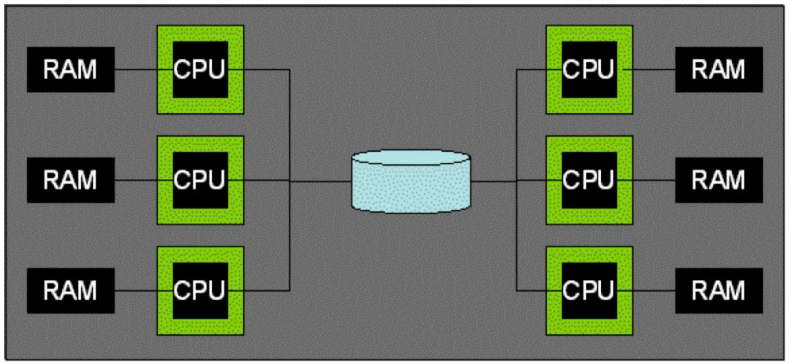
12. Parallel Query Processing

Parallel Architectures

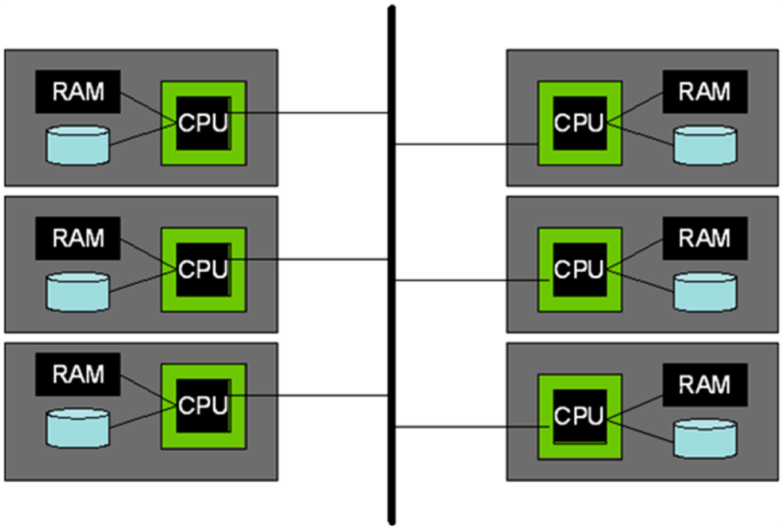
Shared Memory



Shared Disk

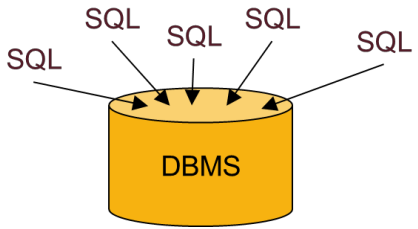


Shared Nothing
(cluster)

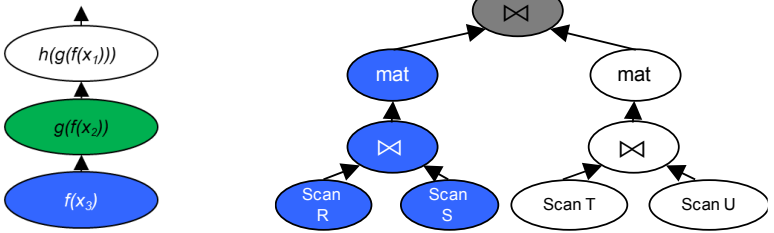


Summary: Kinds of Parallelism

- Inter-Query

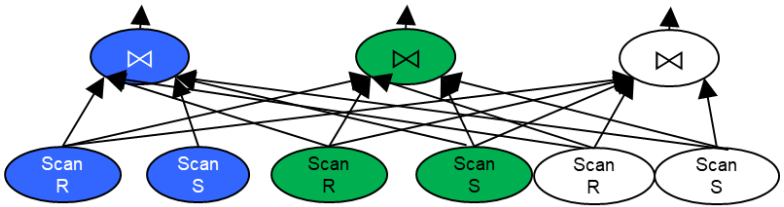


- Intra-Query
  - Inter-Operator



Pipeline Parallelism

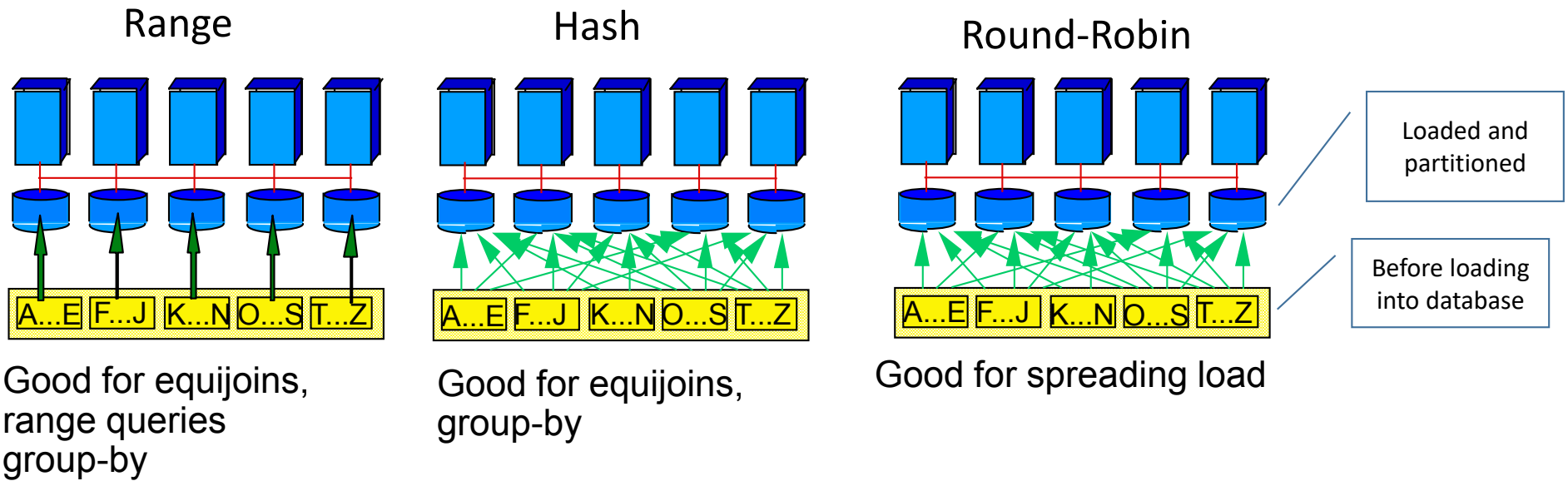
  - Intra-Operator (partitioned)



Partition Parallelism

Data Partitioning

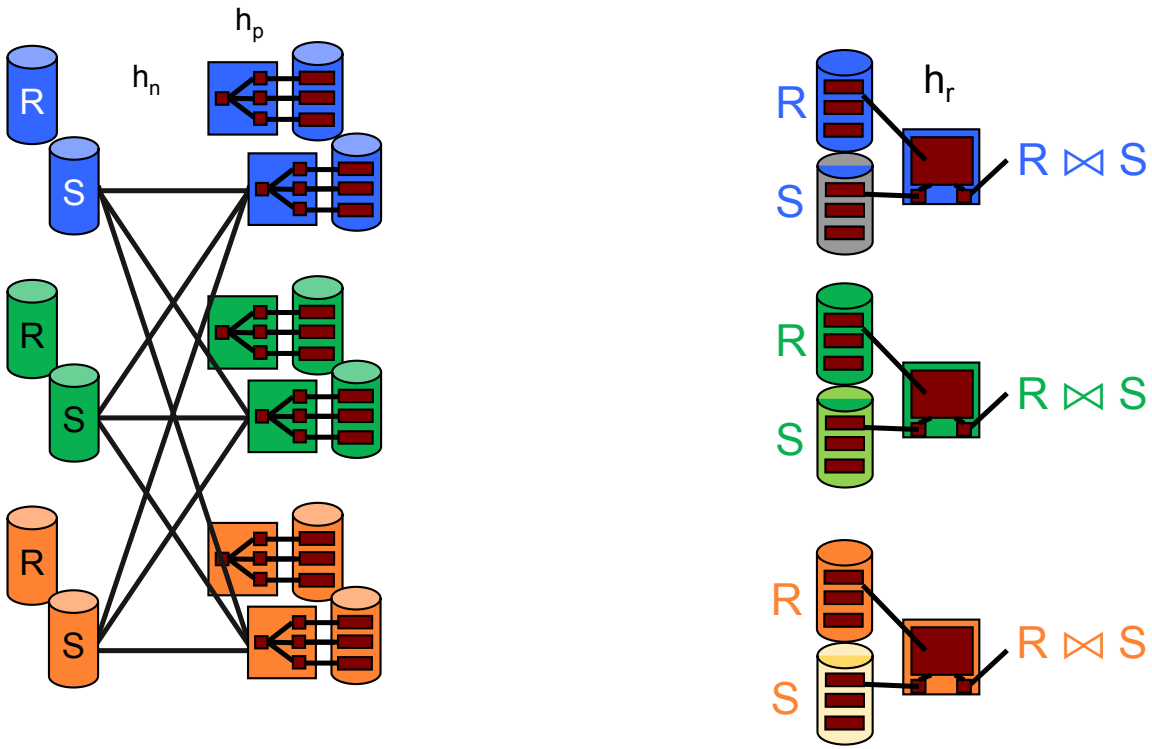
- How to partition a table across disks/machines
  - A bit like coarse-grained indexing!



- Shared nothing particularly benefits from "good" partitioning

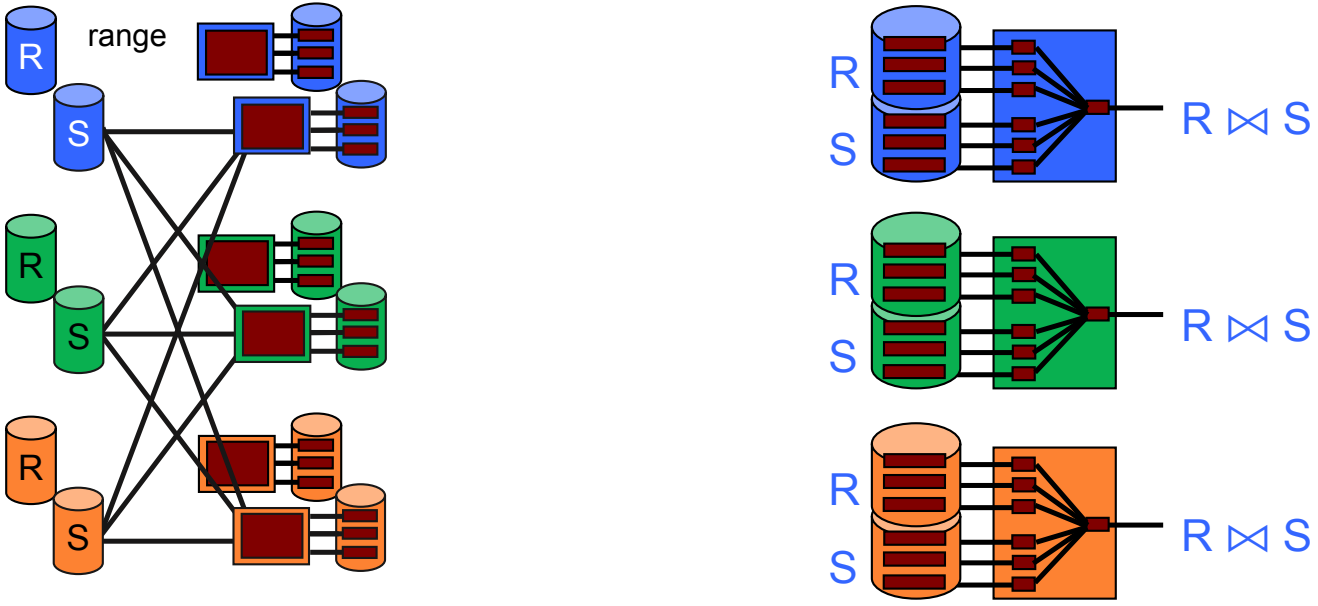
Parallel Grace Hash Join

- Pass 2 is local Grace Hash Join per node
  - Complete independence across nodes



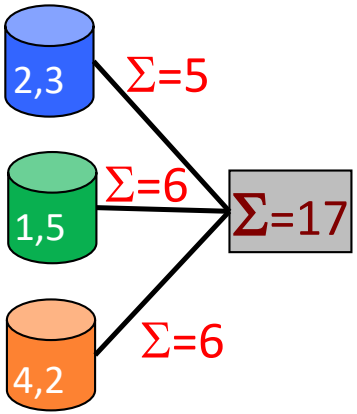
Parallel Sort-Merge Join (with optimization)

- Pass 0 .. n-1 are like parallel sorting above
  - But do it 2x: once for each relation, with same ranges
- Pass n: merge join partitions locally on each node



Parallel Aggregates

- Hierarchical aggregation
- For each aggregate function, need a global/local decomposition:
  - **sum**(S) = Σ Σ (s)
  - **count** = Σ **count** (s)
  - **avg**(S) = (Σ Σ (s)) / Σ **count** (s)
  - etc...



15. MapReduce & Spark

Distributed File System (DFS)

- For very large files: TBs, PBs
- Each file is partitioned into *chunks*, typically 64MB
- Each chunk is replicated several times (≥3), on different racks, for fault tolerance
- Implementations:
  - Google's DFS: GFS, proprietary
  - Hadoop's DFS: HDFS, open source

Typical Problems Solved by MR

- Read a lot of data
- Map: extract something you care about from each record
- Shuffle and Sort
- Reduce: aggregate, summarize, filter, transform
- Write the results

Paradigm stays the same,
change map and reduce
functions for different problems

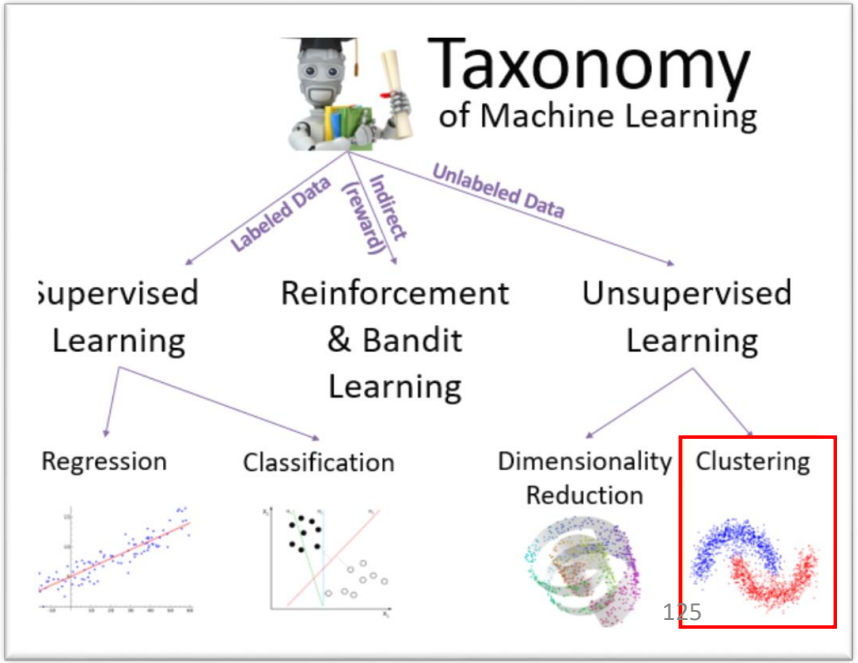
- **Fault handling**: MapReduce handles fault tolerance by writing intermediate files to disk
- **Stragglers**: pre-emptive backup execution of the last few remaining in-progress tasks

Relational Operators in MapReduce

Given relations R(A,B) and S(B,C) compute:

- Selection: $\sigma_{A=123}(R)$
- Group-by: $\gamma_{A,sum(B)}(R)$
- Join: $R \bowtie S$

16.
Data Mining and Machine Learning
Part II -- Clustering



K-Means Algorithm: **Details**

```
centers ← pick k initial Centers

while (centers are changing) {
   // Compute the assignments (E-Step)
   asg ← [(x, nearest(centers, x)) for x in data]

   // Compute the new centers (M-Step)
   for i in range(k):
      centers[i] =
         mean([x for (x, c) in asg if c == i])
}
```

Compute the "Expected" Assignment

Find centers that maximize the data "likelihood"

Guaranteed to converge!

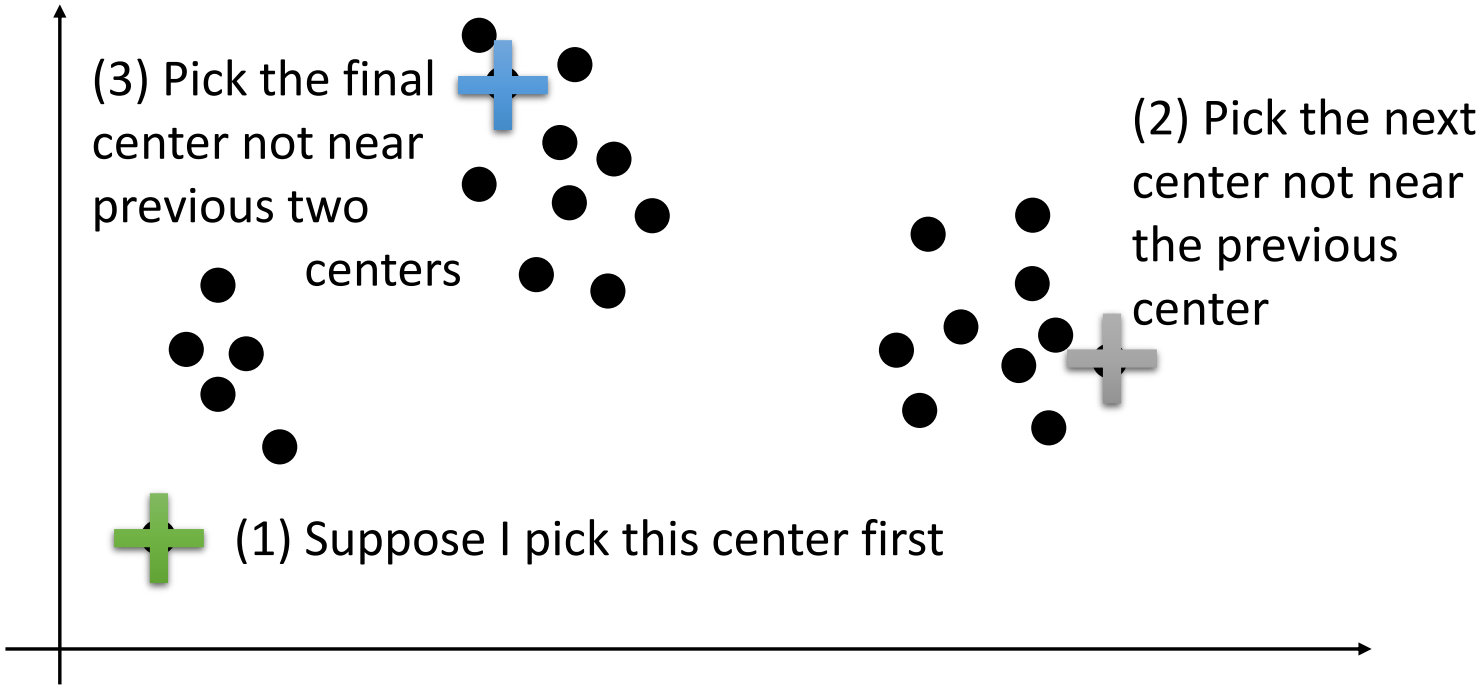
… to what?

To a **local optimum.** ☹

Depends on Initial Centers

Picking the Initial Centers

- **Better Strategy:** kmeans++
  - Randomized approx. algorithm
  - Intuition select points that are not near existing centers



K-Means in Map-Reduce

- **MapFunction**(*old_centers, x*)
  - Compute the index of the nearest old center
  - Return (**key** = *nearest_centers*, **value** = (*x*, 1))
- **ReduceFunction** combines values and counts
  - For each cluster center (Group By)
- Data system returns aggregate statistics:

$$s_i = \sum_{x \in \text{Cluster } i} x_i \qquad \text{and} \qquad n_i = \sum_{x \in \text{Cluster } i} 1$$

- ML algorithm computes new centers: $\mu_i = s_i / n_i$