CS 186: Databases Lecture Notes

Week 1: Lecture 1 Intro (8/24)

Why Database

- How to develop systems to efficiently manage, maintain, process, query, transact with, amek sense of data

#1: Utility

- Systems are incredibly useful

#2: Centralization

- Data centralized

#3: Core of computing

- Abstraction, representation, modeling, reuse, rapid access, declaritivity

#4 Tons of opportunities in academic research

- Turing award

What aspects do we want to worry about?

- Efficiency
- Deal with lots of data
- Don't lose information
- Allow multiple users
- Consistent
- Easy to use

"Database System" Approach: Abstraction

Database System

- System for efficient, convenient, safe, multi user storage of and access to massive amounts of persistent data
- Must rely on stable storage

Relational Model and SQL Basics

- Database: Set of named relations
- Database Management System (DBMS)
- Relation (aka Table)
    - Schema: description ("metadata")
    - Instance: Set of data satisfying the schema
- Attribute (column, field)
- Tuple (record, row)
- Cardinality (# tupics in a row)

Table Implementation

- Tuple row major
- Unique names for columns
- Columns must be primitives

First Normal Form

- All relations must be flat, relation is in first normal form
    - Add products manufactred by each company

SQL Pro Cons

- Declarative language
- Many different implementations
- Two sublanguages
    - DDL - Data Definition Language
        - Define and modify schema
    - DML - Data Manipulation Language
        - Queries can be written intuitively

Primary Key columns

- Provides unique lookup key for the relation
- Cannot have duplicate values
- Made up of >1 column

Week 2: Lecture 2 SQL Overview (8/29)

Primary Key can span multiple columns

Foreign Key

- Key that belongs to another table
- Key must be key in the other table

SQL DML (Data Manipulation Language)

- Find all 27 year old sailors
- Basic Single Table Queries
    - ```
      SELECT [DISTINCT] <column expression list>
      FROM <single table>
      [WHERE <predicate>]
      ```
-
- DISTINCT removes all unique pairs with duplicate rows before output

Ordering

- ORDER BY clause specifies output to be sorted
    - Numeric ordering is numberlike
    - Lexicographically ordering
- Must refer to columns in the output can name a column as an asials

Setting Limits

- LIMIT clause number

Group By

- Partition table into groups with the same GROUP BY column values
    - Can grou by a list of a columns
- Product aggregate results per group = cardinality of output = # distinct gorup values

Having
      ```

- HAVING predicate filters groups applied after group and aggregation
- Can only be used for aggregate questions- Pcoerss

Conceptual Order of Evaluation

```
SELECT [DISTINCT] AVG(S.gpa), S.dept
FROM Students S
GROUP BY S.dept
HAVING COUNT(*) > 2
```

- The HAVING predicate filters groups
- HAVING is applied *after* grouping and aggregation
  - Hence can contain anything that could go in the SELECT list

Conceptual Order of Evaluation

(5) **SELECT** [**DISTINCT**] <*col exp. list*>
(1) **FROM** <*single table*>
(2) [**WHERE** <*predicate*>]
(3) [**GROUP BY** <*column list*>
(4) [**HAVING** <*predicate*>]]
(6) [**ORDER BY** <*column list*>]
(7) [**LIMIT** <integer>];

1.

More Alias: Self-joins

- Select as sname1
- Form cartesian crocc product of 2 tables, generate gigantic cartesian produc
- Filterby criteria

String Comparison

- Any single char

Old School SQL

```
SELECT S.sname
FROM   Sailors S
WHERE  S.sname LIKE 'B_%'
```

_ = any single char; % = zero or more chars
Returns Bob

- %: zer or more chars
- SQLite ~ not supported

Bolean connectedness

- So far: Basic Single-Table DML queries
  - SELECT (with DISTINCT)/FROM/WHERE
  - Aggregation: GROUP BY, HAVING
  - Presentation: ORDER BY, LIMIT
- Extending basic SELECT/FROM/WHERE
  - Multi-table queries: JOINs
  - Aliasing in FROM and SELECT
  - Expressions in SELECT
  - Expressions, string comparisons, connectives in WHERE
  - **Extended JOINs**
  - The use of NULLS
- Query Composition
  - Set-oriented operations
  - Nested queries
  - Views
  - Common table expressions

Week 2: Lecture 3 Disk Buffer, Files (8/31)

Inner/Natural Joins

- If you don't specify
- INNER JOIN
- NATURAL JOIN
- Natural means Equi-join (identical values) for pairs of attributes with the same name

Left Outer Join

- Returns all matched rows, and *preserves all unmatched rows from the table on the left of the join clause* , use NULLs in fields of non-matching tuples

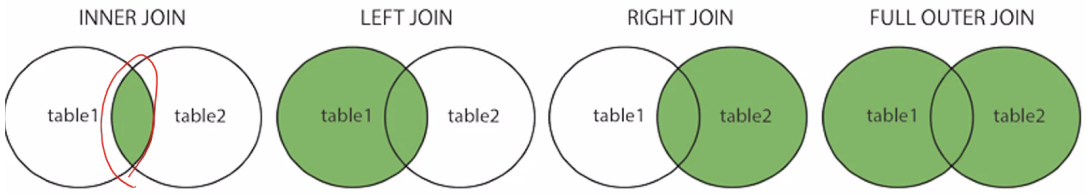
Right Outer Join

- Returns all matched rows, and *preserves all unmatched rows from the table on the right of the join clause* , use NULLs in fields of non-matching tuples

Full OUter Join

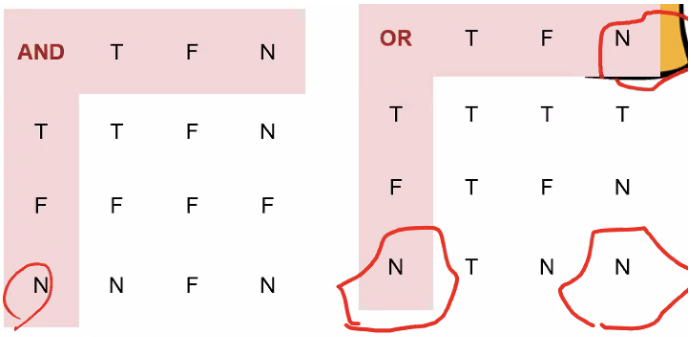
- Returns all (matched or unmatched) rows from the tables on both sides of the join clause

- **(INNER) JOIN**: Returns records that have matching values in both tables
- **LEFT (OUTER) JOIN**: Return all records from the left table, and the matched records from the right table
- **RIGHT (OUTER) JOIN**: Return all records from the right table, and the matched records from the left table
- **FULL (OUTER) JOIN**: Return all records when there is a match in either left or right table



NULL Values

- Values for any data type can be NULL
  - Indicates the value is present but unknown or inapplicable
  - Comes naturally from outer joins
  - Can check by IS NULL$



AND	T	F	N
T	T	F	N
F	F	F	F
N	N	F	N

OR	T	F	N
T	T	T	T
F	T	F	N
N	T	N	N

es

:ing <= 8;

NOT	T	F	N
	F	T	N

-
  - 3 value truth tables
  - NULL values are treated as "I don't know" can either be true or false

NULL and Aggregation

- Typically ignored by aggregate functions

```
SELECT sum(rating) FROM sailors; // sum of non-NULL ratings

SELECT avg(rating) FROM sailors; // avg of non-NULL ratings

SELECT count(rating) FROM sailors; // count sailors with non-NULL ratings

SELECT count(*) FROM sailors; // count all sailors!
```

UNION ALL : Multiset Semantics

- UNION reservers w itself
- Union w itself for reserve will remove duplicates

Sid of sailes who reserved a red AND a green both

- Cannot be both colors

Nested Queries with Correlation

- Nested

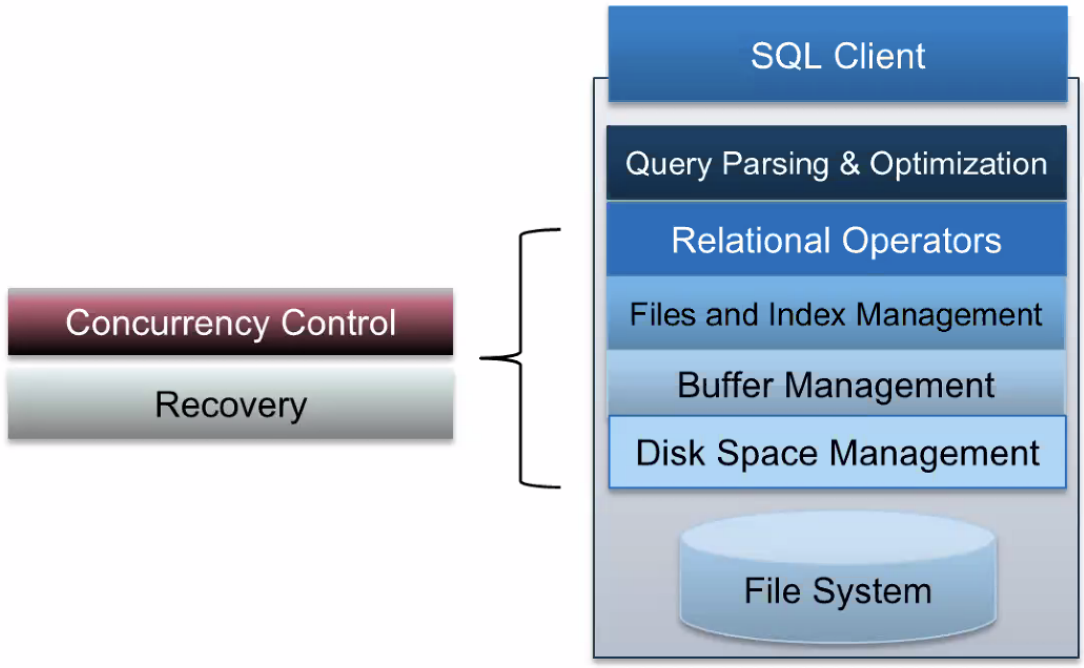
More on set comparison operators NOT, IN NOT EXISTS, op ANY, op ALL

Views Instead of Relations in Queries

- Create View

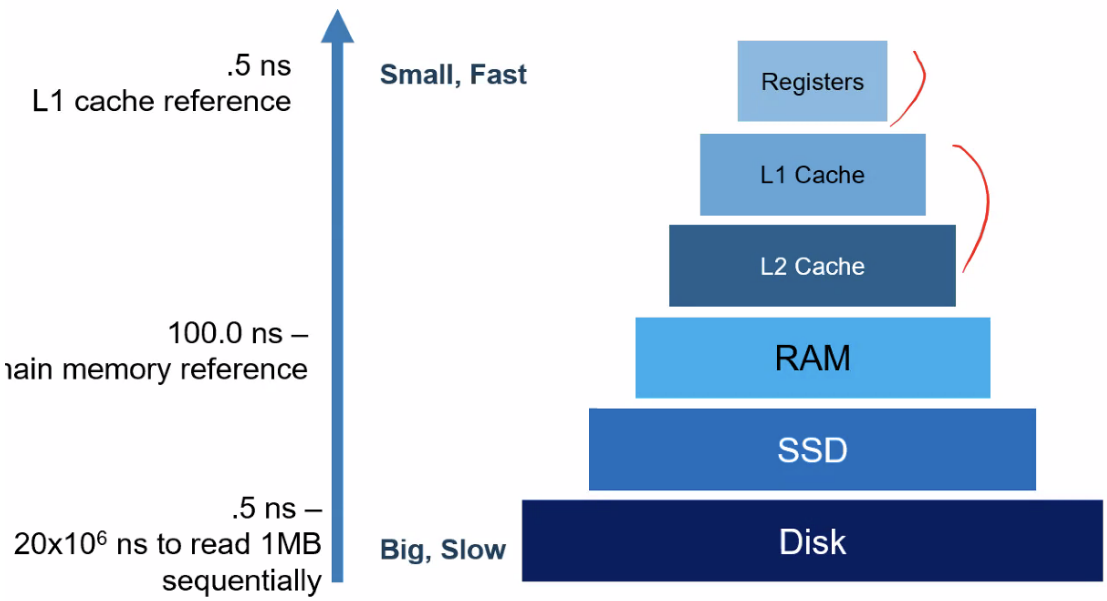
WITH aka common table expression (CTE)
Testing SQL Qeries
- Typically not every database instance will reveal every bug in query
- data base instance without any rows in it

Arcitecutre of a DBMS
- Organized in layers
- Each layers



-

Disks
- Most database systems were originally designed for magnetic spinning disks
- Instilled design dieas that apply to using solid state disks as well
- Disk API
    - READ, WRITE: RAM to disk, disk to RAM



Platters and arms, playhead

Maximize use of storage

Disk Space Managment: Implementation

- Run over our own filesystem
- Bypass OS allocate singular large file on an empty disk assume sequential byte accessa are fast
- Most FS optimize disk layout for sequential access

Week 3: Lecture 4 Cost Models and Indexes (9/7)

Disk Representations

- Files: Pages
    - Each Table is stored in one or more OS Files
    - Each file contains many pages
    - Each page contains contains many records

Files of Pages of Records

- DB File: collection of pages, each containing a collection of records
- API for higher layers of DBMS
    - Reads: Fetch record by record id

Many DB file structures

- Unordered Heap files
- Clustered heap files
- Sorted files

Unordered Heap Files

- Collection of records in no particular order
- As file shrinks/grow, pages allocated or deallocated

Take 1: Heap file as list

- Special header page
    - Location of heap file and header page saved
- Each page contains 2 pointser plus free space and data

Take 2: Use a page directory

- Directory with multiple header pages

Summary

- Table encoded as files which are collection of pages
- Page directory provides loc of pages and free spaces

<u>Page Basics</u>

The Header

- Metadata about the page

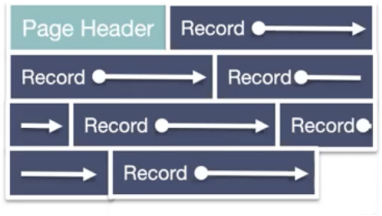
Fixed Length Records

- Packed: pack records densely
    - Reoganized the page when there is a hole

Pack records densely

Record id?

- (pageId, record number in page)
- We know the offset from start of page!
    - Offset = header + (record size) x (n-1)



- Unpacked
- Have bitmap to keep track of slots with records

Variable length records

- Cannot computer offset for record
- Slotted Page: Store the footer stores the pointer to each record since we adda new pointer for each new record
- Lockdown the page when updating
- How many slots in flotted page: growing inward from path to end

Slotted Page Summary

- Good for variable and fixed length records

Record Formats

- Each record in a table/relation has a fixed combo of types
- Record formats what happens if field are variable length: store with padding, account when larger image
- Variable length, can find the length using pointers a record holder

Heap Files

- Unordered collection of records
- API for higher layers of the DBMS, only READ and WRITE
- How? At what cost
    - insert/db

- Suitable when typically access is a full scan of all records
- Sorted files: best ofor retrieval in order, when range of records is needed
- Clusterd files & indexes: grouped together

Cost Model overview

- B: Number of data blocks in the file
- R: Number of records per block
- D: (Average) time to read/write disk block
- Focus: Avg case analysis for uniform random workloads
- Assumptions: we will ignore
  - Sequential vs Random I/O
  - Pre-fetching and cache eviction costs
  - Any CPU costs after fetching data into memory
  - Reading/writing of header pages for heap files
  - Single record insert and delete
  - Equality Selection – exactly one match
- Data need to be brought into memory before operated on

Scan all records

- B * R

Week 4: Lecture 5 B+ Trees (9/12)

Sorted File

- Can use binary search
- Can use my mom

Inserting

- Reading and writing is not free
- For inserting we first need to read the page for the last page and write 2

Delete

- Delete and write back the page need to B/2 + 1
- For sorted file, need to find the page and then shift everything

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

B: Number of data blocks
R: Number of records per block
D: Average time to read/write disk block

Data structures in memory

Index

- Index is data structure that enables fast lookup and modification of data entries by search key
  - Data entries: items stored in the index

    - A pair (k, recordid)
        - Pointers to records in heap files
        - Easy to generalize later

B+ Tree,

- Dynamic Tree Index
    - Always balanced
    - High fanout
    - Efficient insertion & deletion
- +? B-Tree that sotres data entires in leave only
- Where each values are in between the values

Properties

1. Nodes in a B+ tree must obey an occupancy invariant
    a. Each interior node is full beyond a certain minimum
2. Leaves need not be stores in stored order
    a. Help examining them in sequence

Procedure

- Find split on each node
- Follow pointer to next node

For insert when full,

Create a new node and grow the tree into a new height

1. Find the correct leaf L
2. Put data into L
    a. If L space done
    b. Else, must split L
        i. Redistributes entries evenly, copy up middle key
        ii. Insert index entry pointing to L2 into parent in L

Splits grow tree

- Tree growth, gets wider if possible from bottom up
- Worst case

Bulk Loading of B+ Tree

- Keep adding into the leaves and and fill leaf pages to some fill factor , each tree is no longer touched

Note 4: B+ Trees

Properties

- Order of B+ tree: d, d <= x <= 2d entries where entries are sorted
- Pointer to a child node in between each entry of an inner node, at most 2d + 1 child pointers (fanout)
- Keys in left must be less than entry, keys in right are greater than or equal to entry
- All leaves are same depth and have between d and 2d entires

- Every root to leaf path ahs the same number of edges – height of the tree where B+ trees are always balanced
- Only the leaf contains records, inner nodes do not contain actual records

Insertion

1. Find leaf node L to insert in value, traverse down the tree, add the key and record to teh leaf node in order
2. If L overflows (more than 2d entries)
    a. Split into L1, L2, keep d entries in L1, d+1 entires in L2
    b. If L was leaf node, copy L2 first entry into parent, if not leaf, move L2 First entry into parent
    c. Adjust pointers
3. If parent overflows, recurse by doing step 2 on the parent

Inner nodes stores deleted and non-deleted keys

- Alternative-1 indices store the actual records in their leaves.
- Alternative-2 indices store (key, record id) pairs in their leaves.
- Alternative-3 indices store (key, list of record ids) pairs in their leaves.

Week 4: Lecture 6 B+ Trees (9/12)

Week 5: Lecture 7 Buffer Management (9/19)

Handling Dirty Page

- Dirty bit on page
- What to do with a dirty page
    - Write back with a page manager

Advanced Questions

- Concurrent operations,

Buffer Manager

- Buffer pool: large range of memory, malloc'ed at DBMS server boot time (MBs-GBs)
    - Frames
- Buffer manager metadata: smallish array in memory, malloced at runtime

Buffer Manager

Buffer Pool

Buffer Manager metadata

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

When a page is requested

- If requested page is not in pool
    - Choose an un-pinned frame for replacement
    - If frame “dirty”, write current page to disk, mark “clean”
    - Read requested page into frame
- Pin the page and return its address

If requests can be predicted pages can be pre-fetched

- Several pages at the time

Page Replacement Policies

- Two policies discuss:
    - LRU, Clock
    - Most recently used

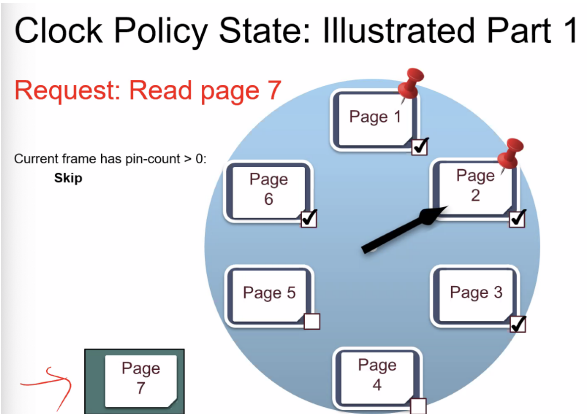
LRU Replacement Policy

- Good for accesses to popular pages
- Can be costly to find min on the last used attribute (priority heap data structure)

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

-

- Clockwise and check each page , if no pin, and reference bit approx last time that the last time the page was used, has been recently used, do not evict but clear the reference bit, not recently used



Repeated Scan (LRU): Read Page 6
Sequentail Scanning + LRU
- Sequential flooding
- 0% hit rate in cache
- Very common in data base worklaods, nested-loops join

Most Recently used policy
LRU wins for random acess
- Hot vs cold
- MRU wins for repeated sequential
- Different heuristics

Why sort
- Rendezvous
- Eleminitaing duplicates
- Explicitly requested: ordering
    - For orderedo utputs

Out-of-core algirhtms
- Twot memes

Better: Double Buffering pt 1
- Main thread runs f(x) on on pair I/O buffers
- SecondI/O Thread drains/fills unused I/O buffers in parallel
- Usable i nany of subsequent discussion

General External MErge Sort
- When more than 3 buffer pages
- How to utlize
- Big matches in pass 0, many streams in emerge passes

Note 5: Buffer Management

Introduction

- Interface between two levels on DBMS: buffer manager
- Buffer manager is responsible for the eviction policy: which pages to evict when space is filled up

Buffer Pool

- Memory partitions the psace into frames that pages can be placed in i

Frame ID	Page ID	Dirty Bit	Pin Count
0	5	1	3
1	3	0	1
2	10	1	0
3			

- Allocates additional space for a metadata table
- Pages are inside frames
- Tracks

1. Frame ID: associated with a memory address
2. Page ID: determines which page a frame currently contains
3. Dirty Bit: whether or not a page has been modified
4. Pin Count: tracks number of requestors currently using a page

Handling Page Requests

- When pages are requested from the buffer manager
    - Page already exists within memory:
        - page pin count is incremented
        - page's memory address is returned
    - Page does not exist in the buffer pool and there is space: next empty frame is found and page is read into that frame
        - Pin count = 1
        - Memory address returned
    - Choice of replacement policy dependent on page access patters and can be chosen by counting **page hits**: when requested page can be found in memory without going to disk
    - **Page miss:** incurs additional IO cost
    - **Hit Rate**: # page hits / (# page hits + # page misses)

LRU Replacement and Clock policy

- **LRU (least Recently Used)**: least recently used unpinned page which has pin count = 0 and lowest value in the Last Used column is evicted, last used column is added to metadata table
    - Costly to implement

Frame ID	Page ID	Dirty Bit	Pin Count	Last Used
0	5	1	3	20
1	3	0	1	32
2	10	1	0	40
3	6	0	0	25
4	1	0	1	15

- **Clock Policy:** alternative that efficiently approximates LRU using a ref bit column in the metadata table and clock hand to track current frame in consideration

Frame ID	Page ID	Dirty Bit	Pin Count	Ref Bit
0	5	1	3	1
1	3	0	1	1
2	10	1	0	1
3	6	0	0	1
4	1	0	1	0

Clock Hand
2

    -
- Sets the clock to the first unpinned frame upon start and sets ref bit to 1 when initially read into a frame
- Eviction
    - Iterate through frames within table, skipping pinned pages and wrapping around to frame 0, until the first unpinned frame with ref bit = 0 is found
    - During iteration, if current frame ref bit = 1, set ref bit to 0 and move to next frame
    - When frame with ref bit = 0, evict page and write to disk if dirty bit = 1, set dirty bit to 0, read in new page, set frame's ref bit to 1, move clock to next frame

- If accessing page currently in the buffer pool, clock policy sets the page's ref bit to 1 without moving the clock hand

Sequential Scanning Performance - LRU

- **Sequential Flooding:** Performance suffers when a set of pages S where |S| > buffer pool sidze are access multiple times repeatedly

MRU Replacement

- **MRU (Most Recently Used)** : evict the most recently used unpinned page measured by when the page's pin count was last decremented

Sequential Scanning performance - MRU

- MRU outperforms

Note 6: Sorting

Traditional sorting algorithms require storing all of data in memory, in databases, most of the time, the data will be an order of magnitude larger than memory available

I/O Review

- We only look at number of I/Os an algorithm incurs when analyzing its performance, ignore any caching done by buffer manager

Two Way External Merge Sort

- Conquer first by sorting records on individual pages, then merge the pages together using merge algorithm
- Result of these merges is **sorted runs :** sequence of pages that is sorted
- Merge these sorted runs until we have only one sorted run remaining

Analysis of Two Way Merge

- First pass over data will take $2 * N$ I/O where N is number of data pages
    - Need to read in every page and write back every page after modifying it
- Each pass cuts the amount of sorted runs left in half and dividing the data each tiem mean $2N * (1 + \log_2(N))$
- 2 buffer pages for the **input buffer** and store the output in an **output buffer**
- Total of 3 pages required in each merging pass

Full External sort

- Assume B buffer pages available
- Load B pages and sort them all at once into a single sorted run
- B - 1 Input buffers and merge together B - 1 sorted runs at a time

Analysis of Full External Merge Sort

- Conquering pass produces only N/B sorted runs now, $2N * (1 + \log_{B-1} N/B)$

Note 7: Hashing

Grouping like values, together, want to build several hash tables and concatenate

- First partitioning pass hashes into B - 1 partition, recursively hashing until partition has less than B

Analysis

- Depends on partitions
- I/O: go through passes $(\sum_{i=1}^{m} r_i + w_i) + 2X$
  - r_i: # pages to read in for partitioning pass
  - m : total partitioning passes req
  - w_i: # pages to write in partitioning pass i
  - X : total pages after partition to build hash table
- $r_0 = N, r_i \leq w_i, w_i \geq r_{i+1}, X \geq N$
-

Note 8: Joins

Introduction

- Final write cost is not included in our join cost models

Simple Nested Loop Join

- **Simple nested loop join (SNLJ) :** for each record in R and search for matches in S
  - I/O: $[R] + |R|\,[S]$ where [R] is number of pages in R and |R| is number of records in R

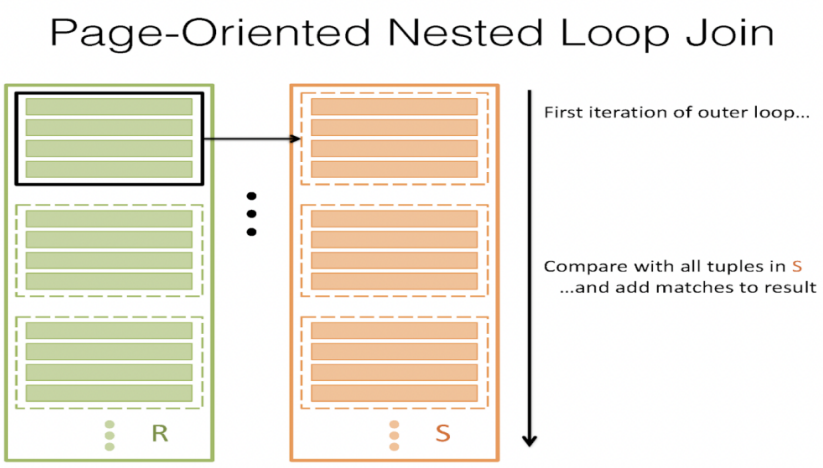
Page Nested Loop Join (PNLJ):

- Read in every single page in S for every single page of R

```
for each page p_r in R:
  for each page p_s in
    For each record r_i in p_r:
      For each record s_j in p_s:
```

```
for each page p_r in R:
    for each page p_s in S:
        for each record r_i in p_r:
            for each record s_j in p_s:
                if θ(r_i, s_j):
                    yield <r_i, s_j>
```

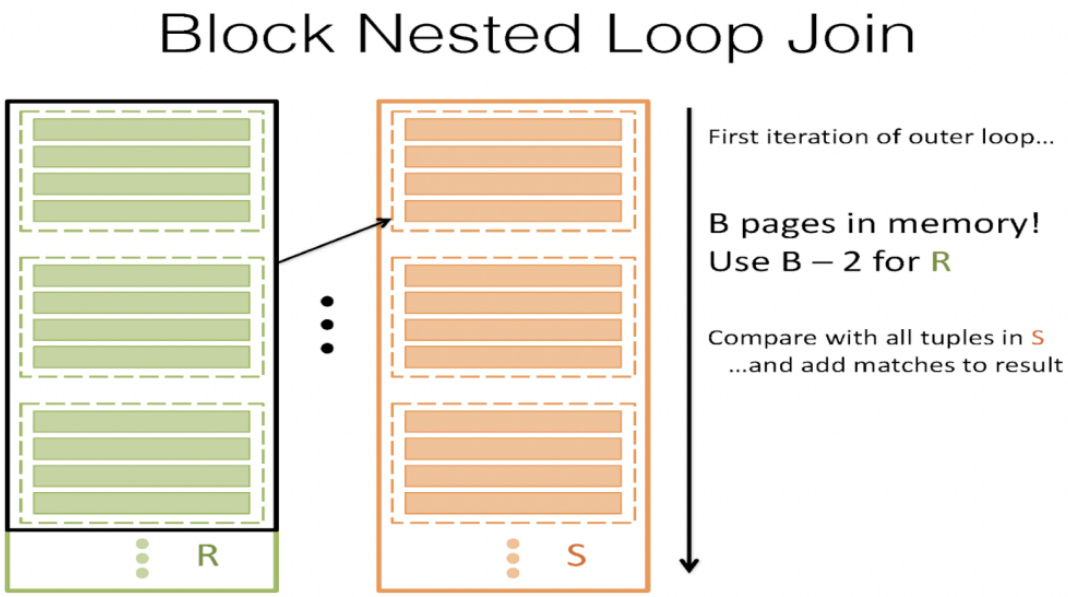
-



-
- I/O: $[R] + [R][S]$

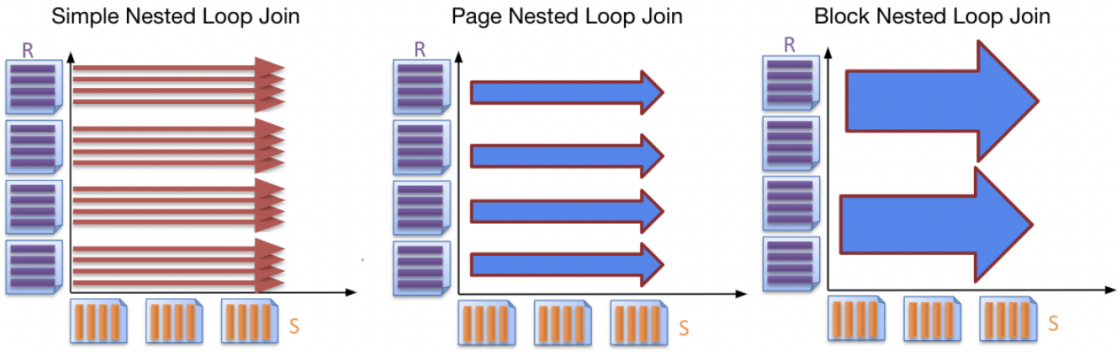
Block Nested Loop Join (Chunk Nested Loop Join)

- B - 2 pages are for R, 1 page is for S, 1 page is output buffer for join
- Utilize the buffer to help reduce the I/O cost to reserve as many pages as possible for a chunk of R



```
for each block of B−2 pages B_r in R:
    for each page p_s in S:
        for each record r_i in B_r:
            for each record s_j in p_s:
                if θ(r_i,s_j):
                    yield <r_i, s_j>
```

-
- For each chunk of R match all records in S against all the recoredsd in the chunk
- I/O: $[R] \ + \ \backslash ceil(\frac{[R]}{B-2})\ [S]$



-

Index Nested Loop Join

- If we have an index on S that is on appropriate field, can look up matches of r in S easier

```
for each record r_i in R:
    for each record s_j in S where θ(r_i,s_j)==true:
        yield <r_i, s_j>
```

The I/O cost is $[R] + |R|*$(cost to look up matching records in S).

-

Naive Hash Join

- Construct a hash table that is B - 2 pages big on records of R, fit into memory, read in each record of S and look up in R's hash table to see if can find matches
- I/O: [R] + [S]

Grace Hash join

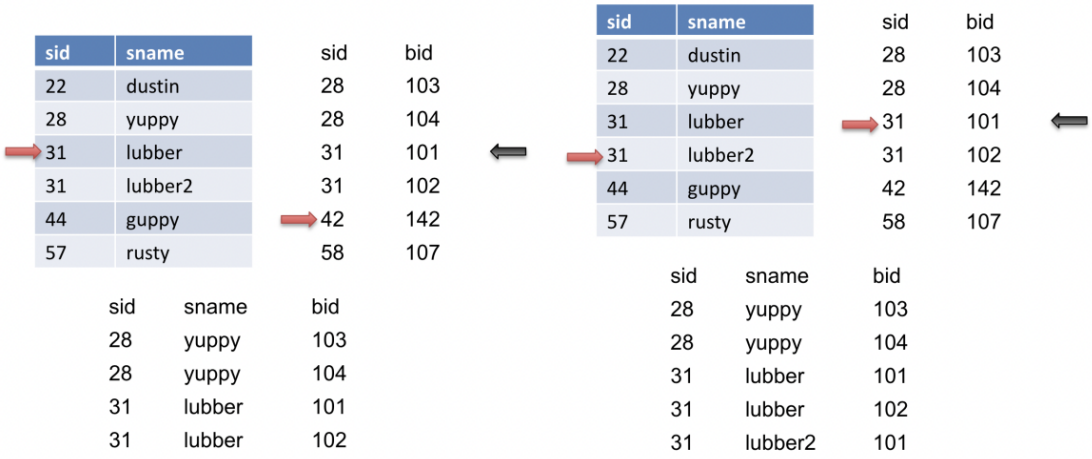
- Hash R and S into B - 1 buffers to get partitions $\leq \ B \ - \ 2$ hash into smaller ones until R or S $\leq \ B \ - \ 2$, stop partitioning and load the smaller partition into memory to get an in-memory hash table and perform naive hash join
- I/O: cost of ahshing plus cost of naive hash join, depends on how many times reepeated hash on partitions
- Sensitive to **key skew:** try to hash but many keys go into the same bucket

Sort Merge Join

- Sort R and S first,

1. Begin at the start of R and S and advance one or the other until we get to a match
2. Assume match, r_i, s_j, mark the spot and check subsequent records in S until we find something not a match
3. Go to next record in R and go back to the marked spot in S and begin at stoep 1

- I/O: cost to sort R + cost to sort S + $[R] + [S]$




```
do {
    if (!mark) {
        while (r < s) { advance r }
        while (r > s) { advance s }
        // mark start of \block" of S
        mark = s
     }
    if (r == s) {
        result = <r, s>
        advance s
        yield result
    }
    else {
        reset s to mark
        advance r
        mark = NULL
    }
}
```

-

- Can combine the last sorting phase with the merging phase

1. Sort $[R]$ `and $[S]$ until both almost sorted
2. How many runs of R and how many runs of S are left, and sum, allocating one page in memory for each run
3. runs(R) + runs(S) $\geq B$, can avoid doing an extra read of both R and S

Note 9: Relational Algebra

Relational Algebra

- Procedural programming language

Relational Algebra Introduction

- Take in a relation and output a relation
- Cannot have duplicates

Projection (π)

- **Project Operator** - takes in single relation and selects the solumns specified
- SELECT name FROM dogs; is the same as $\pi_{name}(dogs)$

Selection (σ)

- Takes in single relation and filters rows based on certain condition
- Similar to WHERE
- SELECT name, age FROM dogs WHERE age = 12 is same as $\sigma_{age = 12}(\pi_{name, age}(dogs))$

Union (∪)

- Combine data from different relations and removes duplicates
- $\pi_{name}(dogs) \cup \pi_{name}(cats)$

Set Different (-)

- Returns every row in the first table except the rows that also show up in the second table, no duplicates,
- Similar to EXCEPT
- $\pi_{name}(dogs) - \pi_{name}(cats)$

Intersection (∩)

- INTERSECT SQL
- $\pi_{name}(dogs) \cap \pi_{name}(cats)$

Cross Product (x)

- Performing a Cartesian Product in SQL one tuple for every possible pair of tuples form both relations
- $dogs\ x\ parks$

Joins ($\backslash join$) ><

- Inner join is left table on the left side, join condition on the subscript
- Not specifying join condition is a natural join

Inner join ($\backslash join_{\theta}$)

- Inner join

Rename (ρ)

- Aliasing
- $\rho_{name -> dname}$

Group By / Aggregation (γ)

- Group by aggregation operator using GROUP BY and HAVING clauses

- SELECT age FROM dogs GROUP BY age HAVING COUNT(*) > 5 as $\gamma_{age,\, COUNT(*)>5}(dogs)$

Note 10: Query Optimization

Databases can change the order they execute the operations in order to get the best performance

Introduction

- Databases can change the order they execute the operations to get the best performance, in terms of I/O
- Query Optimization finds the **query plan** that minimizes the number of I/Os it takes to execute the query, a sequence of operations that will get us the correc tresult for a a query using relational algebra

Iterator Interface

- Generate a sequence of operators that returns the correct output efficiently
- Creates instances of each operator, **iterator interface** which is responsible for efficiently executing operator logic and forwarding relevant uples to the next operator
- Each time next() is called, until the base case, we either
    - **Streaming operators** using a small amoutn of work to produce each tuble or
    - **Blocking operators** like sort which do not product output until they consume their entire input

Selectivity Estimation

- No way of knowing how many I/O a plan will cost until execution
- Cannot gauaranttee optimal, so need a wa y to estimate using heuristics
- **Selectivity estimation:** selectivity of an operation is an approximation for what percentage of pages will make it through the operator
    - **X=a**: 1/(unique vals in X)
    - **X=Y**: 1/max(unique vals in X, unique vals in Y)
    - **X>a**: (max(X) - a) / (max(X) - min(X) + 1)
    - **cond1 AND cond2**: Selectivity(cond1) * Selectivity(cond2)
    -

Selectivity of Joins

- For joining

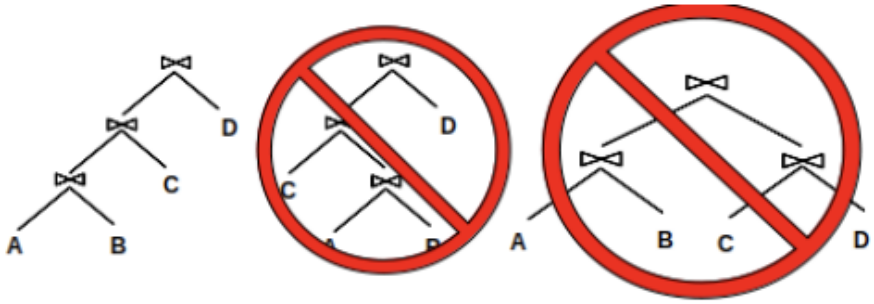
$$\frac{|A|*|B|}{max(unique\ vals\ for\ A.id,\ unique\ vals\ for\ B.id)}$$

To find a join's output size in pages, we need to follow 3 steps:

1. Find the **join's selectivity**. For a join on tables A and B on the condition A.id = B.id, we can refer to the formula above for this.
2. Find the **estimated number of joined tuples** in the output by multiplying the selectivity of the join by the number of joined tuples in the Cartesian Product between the two tables.
3. Find the **estimated number of pages** in the join output by dividing by the estimated number of joined tuples per page.

Common Heuristics

1. Push down projects (π) and selects (σ) as far as they can go
    a. Reduces pages the other operators have to deal with
2. Only consider left deep plans
    a. All right tables in the join are the original tables



    b. Reduces plan space, can be fully pipelined
3. Do not consider cross joins unless they are the only option
    a. Cross ojins produce a ton of pages making it perform many I/Os

Pass 1 of **System R** (**Selinger Optimization**)

- System R: query optimizer using all the heuristics
- How to access tables during the first pass
    - Full Scan
        - Always [P] I/O for table P
    - Index Scan
        - Depends on how records are stored
        - Alternative 1: (cost to reach level above leaf) + (num leaves read)
        - Alternative ⅔: (cost to reach level above leaf) + (num of leaf nodes read) + (num of data pages read)
    - Only return a row if it matches all the single table conditions
    - Which access plans we will advance to subsequent passes to be considered, advance the optimal access with optimal **interesting order** where sorted by a column used in ORDER BY or GROUP BY or downstream join (join not evaluated

yet)

Passes 2...n

- Joining the tables together, join i tables
- Join products a sorted output Sort Moerge Join (SMJ), use SMJ for the last join in the set, sorted on join condition
- Simple Nested Loop Join (SNLJ) and index nested loop join (INLJ) can preserve a sorted ordering on the left relation
- Grace Hash Join (GHJ), Page Nested Loop Join (PNLJ), BLock nested Loop Join (BNLJ) never produce an interesting ordering,

Calculating I/O Costs for Join operations

- We may not directly be able to use the formulas from Iterators/Joins
- Whether we materialize intermediate relations (outputs from preivous operators) or stream tehem into input of the next operator
    - Need to write to disk if we materialize intermediate relations
    - For System R query optimizer, assume we never materailzie the outputs of any intermediate operators and they are streamed in as input to next operator
- If inetreresting orders from previous operators may reduce the I/O costs of join
    - If use index scan, no longer need to run external sorting

Note 11: Transactions and Concurrency I

Problems with concurrency

- Inconsistent Reads (Write-Read Conflict)
    - User reads only part of what was updated
- Lost Update (Write - Write Conflict)
    - Two users try to update the same record at the same time so one of the updates gets lost

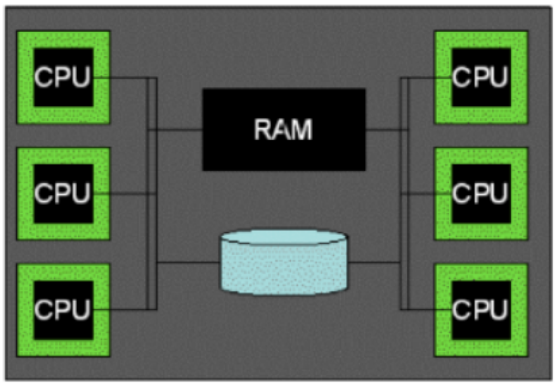
Note 15: Parallel Query Processing

Parallel query processing: query run on multiple machines in parallel

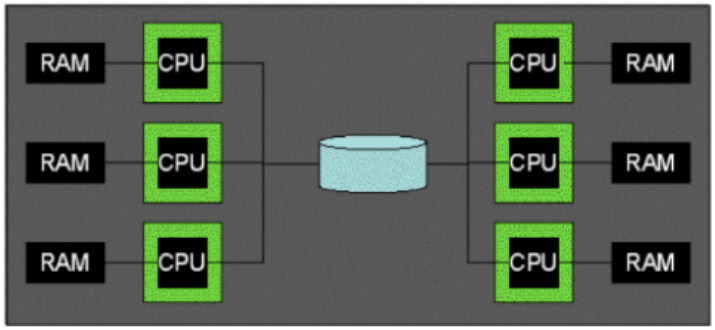
Parallel Architectures

- CPU share memory and disk

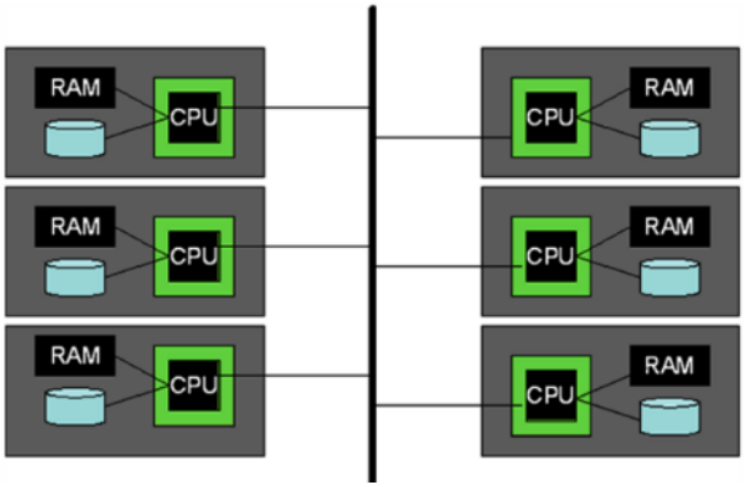
-



- Share disk



- Shared nothing

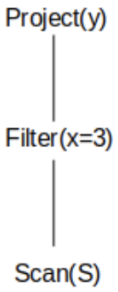


- intra -query parallelism: make one query run as fast as possible by spread the work over multiple computers
- Inter-query parallelism: give each machine different queries to work on to achieve high throughput and complete as many queries as possible

Intra-query parallelism

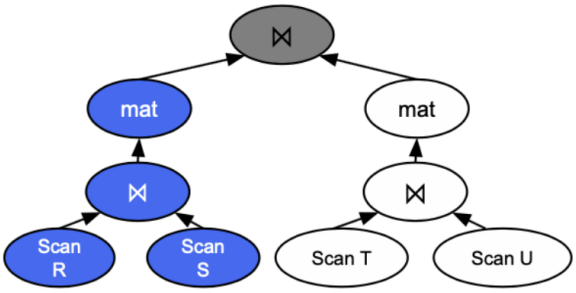
1. Intra-operator: making one operator run as quickly as possible
    a. Divid up data onto several machines and strt in parallel
2. Inter-operator: Making a query run as fast as possible by running operators in parallel

a. Have sort R on one machine and sort S on another machine
b. Pipeline parallelism
    i. Records passed to the parent operator as soon as done
    ii.



c. Bushy Tree parallelism
    i. Different branches of the tree are run in parallel
    ii. Left branch and right branch run at the same time

1.



Partitioning

- Sharding: Each data page stored on only one machine, better performance
- Replication: data page appeared on multiple machines -> better availability
- Partitioning scheme
    - Determine what machine a certain record will end up on

1. Range Paritioning
    a. Each machine gets a certain range of values that it will store
    b. Good for queries that lookup on a specific key, only need to request from machines that the values reside on
    c. Good for parallel sorting and parallel sort merge join
2. Hash partitioning
    a. Each record hashed and sent to a machine
    b. Good for key lookups but not range queries
3. Round Robin Partitioning
    a. Record by record and assign each record to the next machine
    b. Each machine guaranteed the same amount of data
    c. Maximum parallelization , every machine activated for every query

Network cost

- Network cost is how much data we need to send over the network to do an operation,

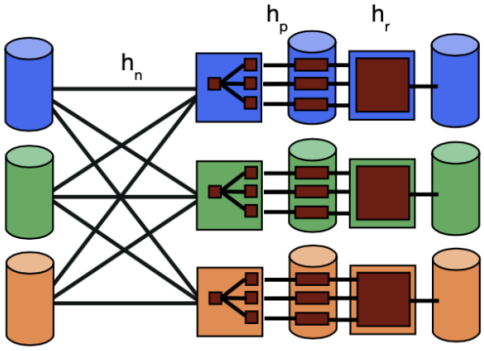
incurred wehenver send data from one machine to another measured in KB

Parallel Sorting

- Range partition the table
- Perform local sort on each machine
- The entire table is then in sorted order
- passes: 1 pass to partition the table across machine + number of passes needed to sort table
    - $1 + \text{ceil}(1 + \log_{B-1} ceil(N/mB)$
        - M is number of machines

Parallel Hashing

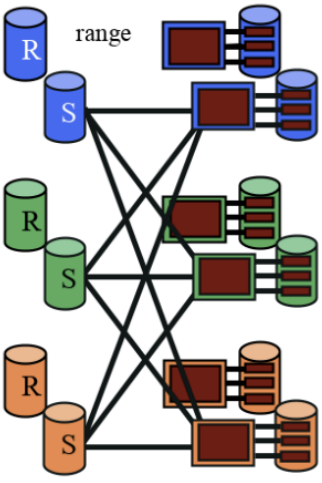
1. Hash Partition the Table
2. Perform local hashing on each machine

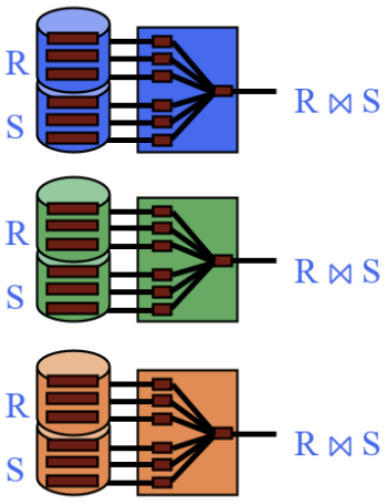
a. 

- Guarantees that like values will be assigned to same machine

Parallel Sort Merge Join

1. Range Parittion each table using the same ranges on the join column
2. Perform local sort merge join on each machine





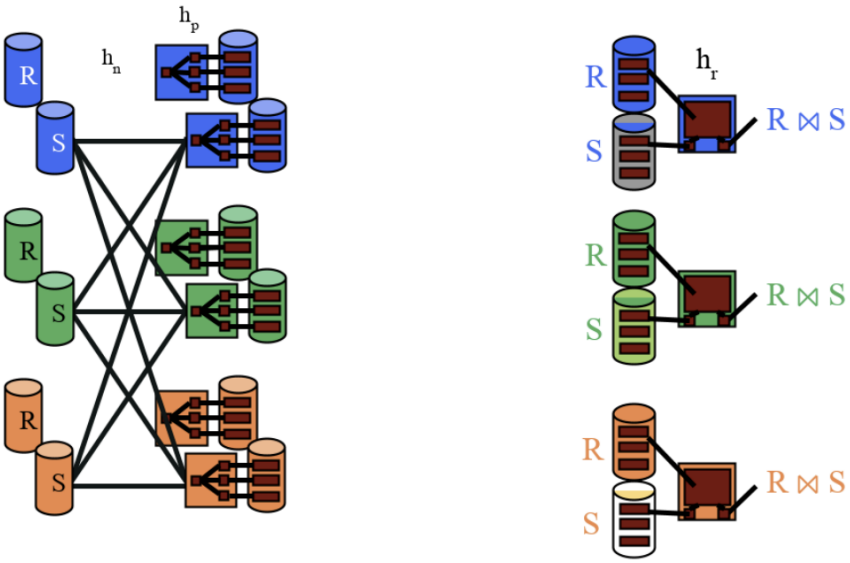
- Passes: 1 pass/table to partition across machines + number of passes needed to sort R +

number of passes to sort S + 1 final merge sort pass, going through booth table s

- Passes: 2 + ceil(1 + $\log_{B-1} ceil(R/mB)$ + $\log_{B-1} ceil(S/mB)$ + 2 passes

Parallel Grace Hash Join

1. Hash Partition each table using the same hash function on join column
2. Perform local grace hash join on each machine

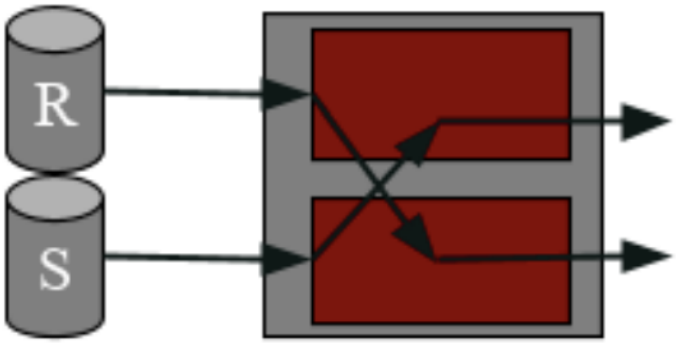


Symmetric Hash Join

- Pipeline breakers: previous join algorithms
- Join algorithms that is pipeline-friendly, produce output as soon as we see our first matches

1. Build two hash tables, one for each table in the join
2. When record from R arrives, probe the hash table for S for all of the matches
3. Whenever a record arrivesa ddit to tis corresponding hash table after probing the other hash table for matches

- Every output tuple gets generated exactly once, when the second record in the match arrives -> produce output when we see a match



-

Hierarchical Aggregation

- Hierarchical aggregation : how we parallelize aggregation operations

COUNT

- Each machine inidivualy counts their records
- All send the counts to the coordinator machine who will sum them together to figure out the overall count
-

AVG

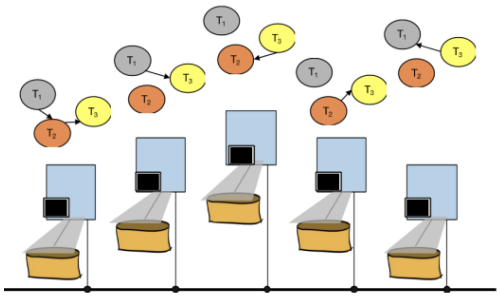
- Each machine must calculate the sum of all the values and the count, send those values to the coordinator machine, adds up the sums and divides by sum of the count

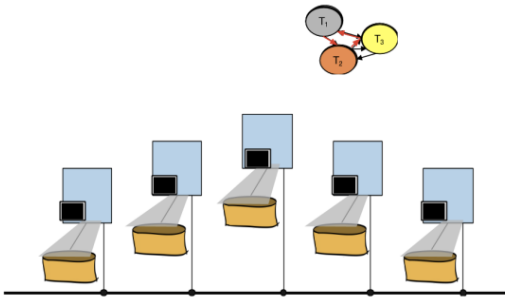
Note 16: Distributed Transactions

Each node receives a partition of the data set through a network

Distirbuted Locking

- Every nock can maintain its own local lock table
- Locking -> 2 phase locking
- In order to find deadlock, draw all waits-for graphs onto top of each oether in a distributed system
-



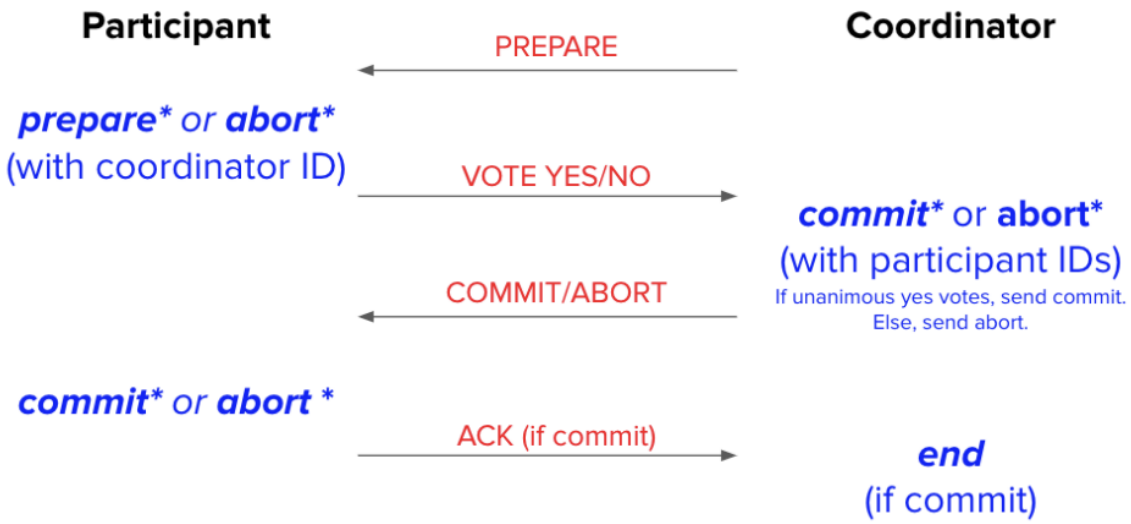


Two Phase Commit (2PC)

- Ensure all nodes reach consensus, all nodes agree on one course of action
- Enforce that all nodes maintain the same view of the data, commit or abort on all nodes involved
- Preparation Phase
    1. Prepare for commit or abort
    2. Prepare or abort record and flush record to disk
    3. Participants send yes vote to coordinator if prepare record is flushed or no vote if abort record is flushed

4. Generates a commit record if it receives unanimous yes votes or an abort record otherwise, flush to disk

- Commit/abort phase
    1. Coordinator broadcasts the result of the commit/abort vote based on flushed record
    2. Participants generate a commit or abort record based on the received vote message an flush record to disk
    3. Participatns send an ACK to coordinator
    4. Coodinator generates an end record once all ACKs are received



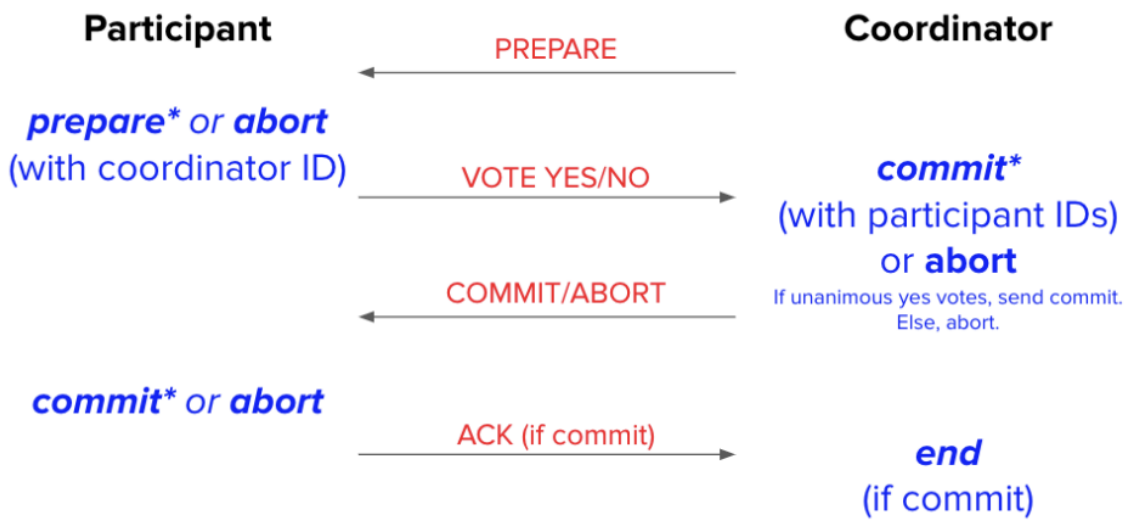
Distributed recovery (2PC)

- If a node fails and comes back online, it should still end up making the same decision as all the other nodes in the database
- Able to figure out from looking at the log

Possible failures

- Participatn is recovering, no prepare record
    - Has not started 2PC, it aborts locally
- Participant is recovering, sees a parepare record
    - Don't know whether or not coordinato rmade a commit decision, ask coordinator , respond with teh commit /abort decision
- Coordinator is recovering, sees no commit record
    - Abort the transaction locally
- Coodinator recovering, sees a commit record
    - Rerun phase 2
- Participant is recovering and ses a commit record
    - Send ack to coordinator
- COordinator is recovering and ses an end record
    - Everybody finished and no recovery

- Abort if we see no log record
- Presumed abort : Abort records never have to be flushed



- Abort records no longer need to be flushed to disk
- Coordinator is recovering and sees an abort record
    - With presumed abort: abort the transaction locally: no messages need to be sent out
    - Without presumed abort: rerun phase 2
- 2PC recovery decision is commit if and only if the coordinator has logged a commit record ,
- Make decision when all nodes are alive