CSE 410: Midterm Review

May 3-6, 2024

Exam Day

- **Do** have...
  - Writing implement (pen or pencil)
  - One note sheet (up to $8\frac{1}{2} \times 11$ inches, double-sided)
- You will **not** need...
  - Computer/Calculator/Watch/etc...

Abstract Disk API

- **Disk** : A collection of **Files**
- **File** : A list of pages, each of size P ($\sim 4K$)
  - `file.read_page(page)`: Get the data on page page of the file.
  - `file.write_page(page, data)`: Write data to page page of the file.

The number of calls to read/write is the IO Complexity

Complexity

```
const RECORDS_PER_PAGE = sizeof::<Record>() / PAGE_SIZE;

fn get_element(file: File, position: u32) -> Record
{
  let page = position / RECORDS_PER_PAGE;
  let data = file.read_page(page);
  return get_records(data)[position % RECORDS_PER_PAGE];
}
```

Complexity

```rust
fn find_element(file: File, key: u32) -> Record
{
  let mut records: Vec<Record> = Vec::new()
  for page in (0..N)
  {
    let data = file.read_page(idx);
    for record in get_records(data)
    {
      records.push(record);
    }
  }
  return records.binary_search(key)
}
```

Streaming Reads/Writes

```
struct BufferedFile {
  file: File,
  buffer: Page,
  page_idx: u32,
  record_idx: u16,
}
impl BufferedFile {
  fn append(&mut self, record: Record) {
    self.buffer[self.record_idx] = record;
    self.record_idx ++;
    if self.record_idx >= RECORDS_PER_PAGE {
      self.file.write_page(self.page_idx, self.buffer);
      self.record_idx = 0; self.page_idx ++;
    }
  }
}
```

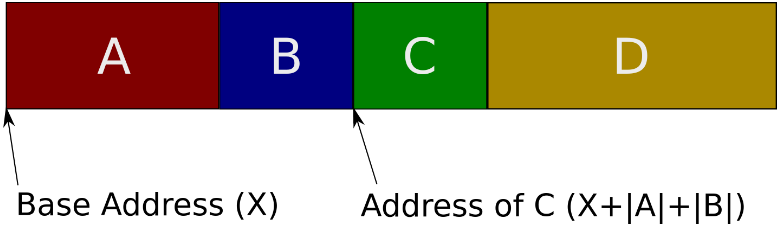
Streaming Reads/Writes

```
struct BufferedFile {
  file: File,
  buffer: Page,
  page_idx: u32,
  record_idx: u16,
}
impl BufferedFile {
  fn next(&mut self) -> Record {
    if self.record_idx >= RECORDS_PER_PAGE {
      self.file.read_page(self.page_idx)
      self.page_idx += 1; self.record_idx = 0
    }
    self.record_idx += 1
    return self.buffer[self.record_idx - 1];
  }
  ...
}
```

Complexity

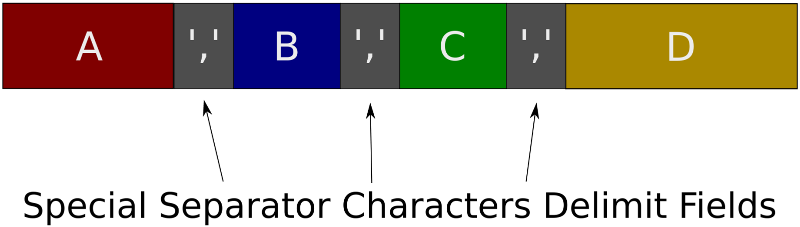
```
fn group_by_sum(input: BufferedFile, output: BufferedFile) {
  let mut buffers: Vec<BufferedFile> = Vec::new();
  for _i in (0..B)  { buffers.push(BufferedFile::new()); }
  while !input.done() {
    let record = input.next();
    let i = HASH(record.key) % B;
    buffers[i].append(record)
  }
  for i in (0..B) {
    let local_sums: Map<String,f32> = Map::new()
    buffer[i].reset()
    while !buffer[i].done() {
      let record = buffer[i].next();
      local_sums[record.key] += record.value;
    }
    for key, value in local_sums {
      output.append( Record { key, value } )
} } }
```

Record Layouts



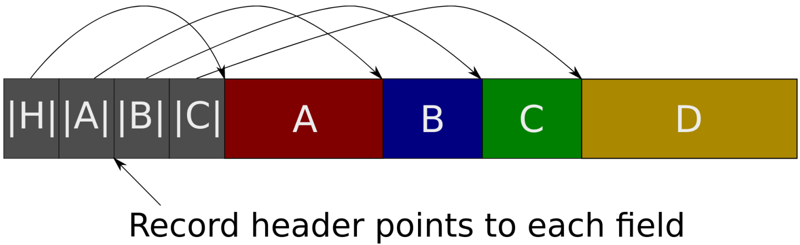
$O(1)$ **field lookup**

Record Layouts



$O(N)$ **field lookup**

Record Layouts

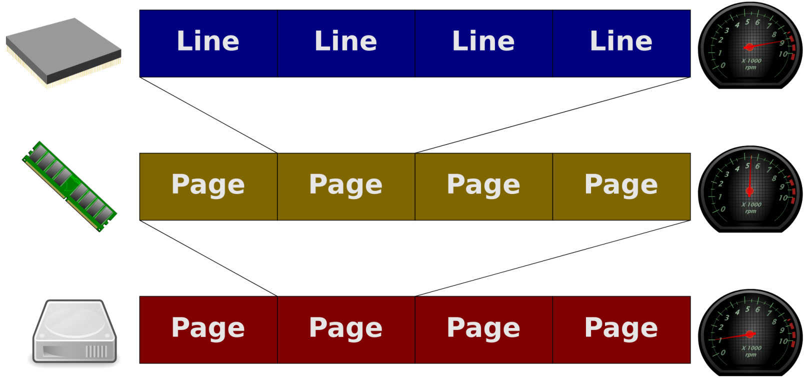


$O(1)$ **field lookup**

Record Layouts

- **Fixed**: Constant-size fields. Field i at byte $\sum_{j<i} |Field_j|$.
- **Delimited**: Special character or string (e.g., ,) between fields.
- **Indexed**: Fixed-size header points to start of each field.

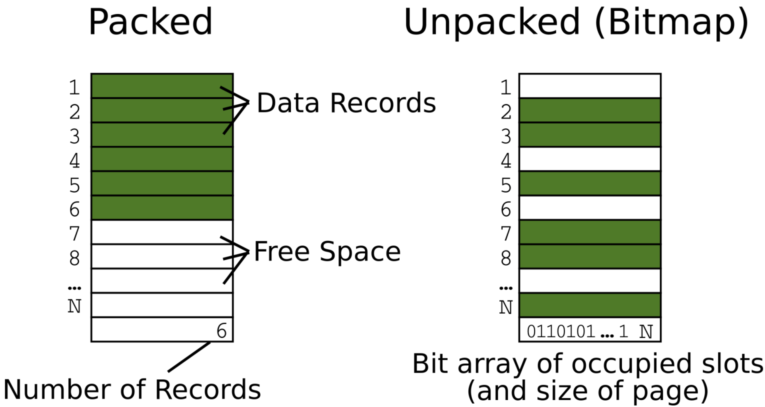
Page Layouts



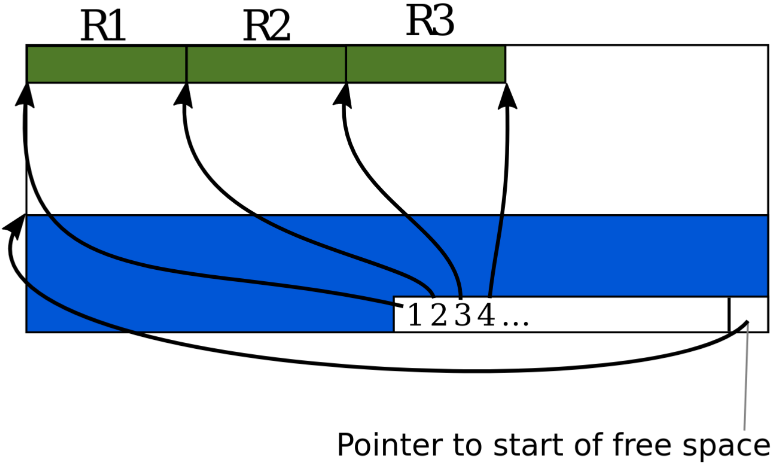
Page Layouts

- **Fixed**: Constant-size records. Record i at byte $i \cdot |Record|$.
- **Delimited**: Special character or string (e.g., `\n`) between records.
- **Indexed**: Fixed-size header points to start of each record.

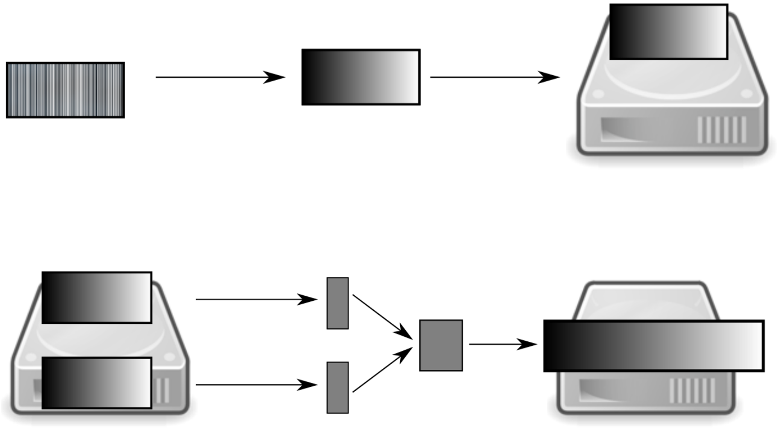
Page Layouts



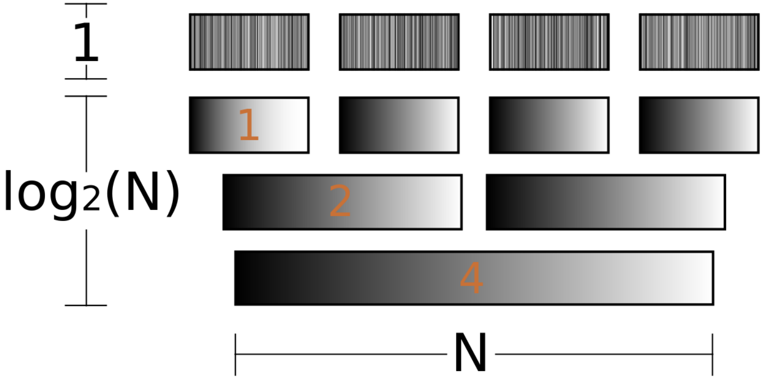
Page Layouts



2-Pass Sort



2-Pass Sort



2-Pass Sort

- **Pass 1**: Use $O(K)$ memory for the initial buffer
- **Pass 2**: Merge $O(K)$ buffers simultaneously

Aggregation

TREE_ID	SPC_COMMON	BORONAME	TREE_DBH

{}

180683	'red maple'	'Queens'	3

{ 'red maple' = 1 }

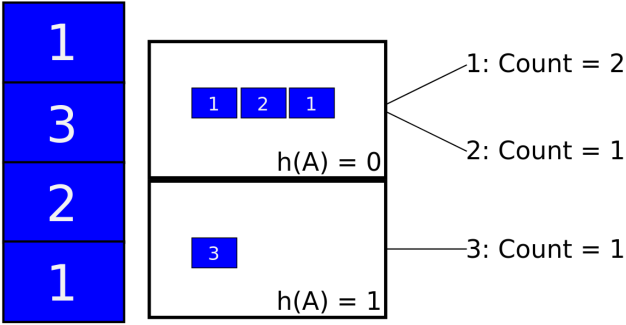
204337	'honeylocust'	'Brooklyn'	10

{ 'red maple' = 1, 'honeylocust' = 1 }

315986	'pin oak'	'Queens'	21

{ 'red maple' = 1, 'honeylocust' = 1, 'pin oak' = 1 }

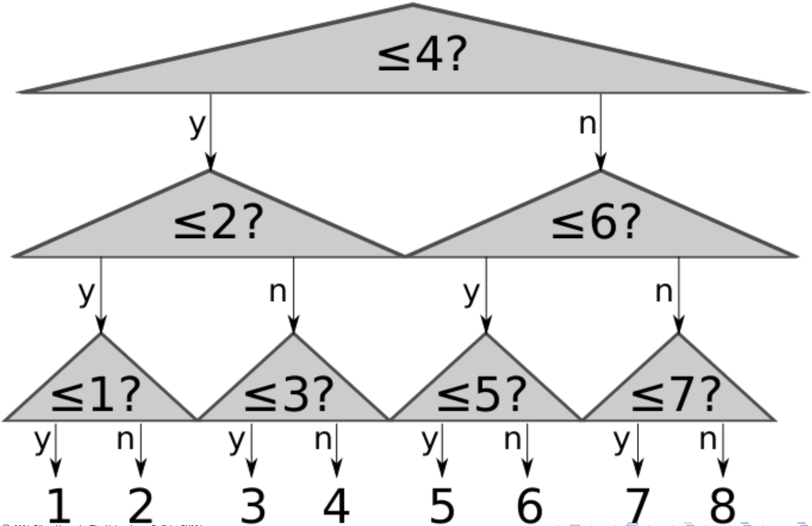
Aggregation



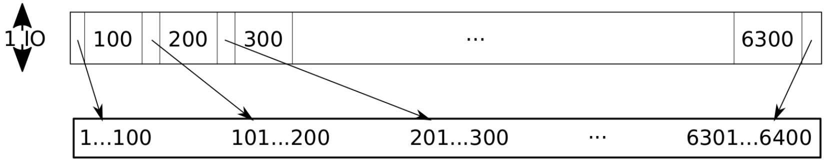
Aggregation

TREE_ID	SPC_COMMON	BORONAME	TREE_DBH
{}			
204337	'honeylocust'	'Brooklyn'	10
{ 'honeylocust' = 1 }			
204026	'honeylocust'	'Brooklyn'	3
{ 'honeylocust' = 2 }			
... and more			
315986	'pin oak'	'Queens'	21
{ 'honeylocust' = 3206, 'pin oak' = 1 }			

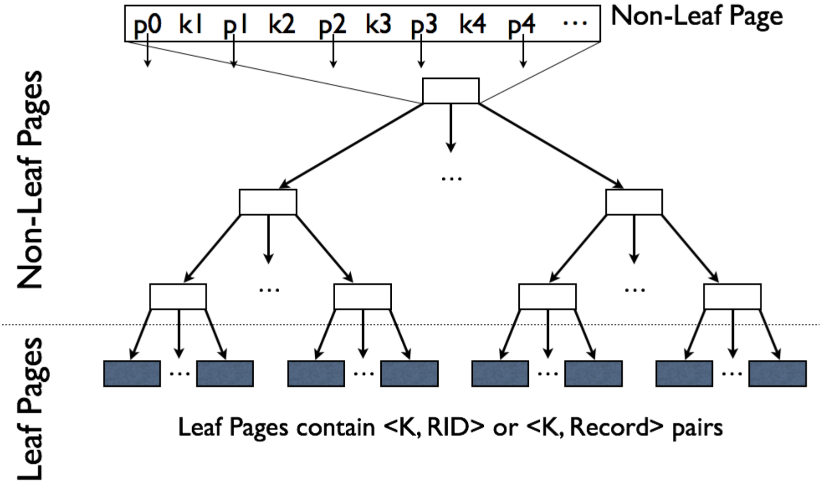
Binary Search



Fence Pointers



ISAM Index


p0 k1 p1 k2 p2 k3 p3 k4 p4 ...
Non-Leaf Page
Non-Leaf Pages
Leaf Pages
Leaf Pages contain <K, RID> or <K, Record> pairs

B+ Tree

Like an ISAM index, but not every page needs to be full, and...
Any page (except the root) must be at least half-full

- Splitting a full page creates a half-full page.
- On deleting the $\frac{P}{2}$th record, steal record from adjacent page.
- If no records can be stolen, must be able to merge with an adjacent page.

B+ Tree

With P records / key+pointer pairs per page:

get(k)

- $O(1)$ Memory complexity
- $O(\log_P(N))$ IO complexity
  - Contrast: $O(\log_2(N))$ in binary search

put(k, v)

- $O(1)$ Memory complexity
- $O(\log_P(N))$ IO complexity
  - $O(\log_P(N))$ reads
  - $O(\log_P(N))$ writes; $O(1)$ amortized writes

LSM Tree

Insight: Updating one record involves many redundant writes in a B+ Tree

Building Block: Sorted Run

- Originally: ISAM Index
- Now: Sorted Array + Fence Pointers (optional Bloom Filter)

LSM Tree

- In-Memory Buffer
- Level 1: B records
- Level 2: $2B$ records
- Level 3: $4B$ records
- Level i: $2^{i+1}B$ records

LSM Tree

put(k,v)

- Append to in-memory buffer.
- If buffer full, sort, and write sorted run to level 1.
- If level 1 already occupied, merge sorted runs and write result to level 2.
- If level 2 already occupied, merge sorted runs and write result to level 3.
- ...
- If level i already occupied, merge sorted runs and write result to level i+1.

LSM Tree

get(k,v)

- Linear scan for k over in-memory buffer.
- If not found, look up k in level 1.
- If not found, look up k in level 2.
- ...

LSM Tree

update(k,v)

- exactly as **put**
- ... but when merging sorted runs, if both input runs contain a key, only keep the newer copy of the record.

delete(k)

- exactly as **update**, but write a 'tombstone' value.
- If **get** encounters a tombstone value, return "not found".
- When merging into lowest level, can delete tombstone.

$\beta - \epsilon$ Trees

Like B+ Tree, but directory pages contain a buffer.

- Writes go to the root page buffer.
- When the root page buffer is full, move its buffered writes to level 2 buffers.
- When a level 2 buffer is full, move its buffered writes to level 3 buffers.
- ...
- When the last directory level buffer is full, apply the writes to the relevant leaves.

Set

- **add(k)**: Updates the set.
- **test(k)**: Returns true iff **add(k)** was called on the set.

Lossy Set

- **add(k)**: Updates the set.
- **test(k)**:
  - Always returns true if **add(k)** was called on the set.
  - Usually returns false if **add(k)** was not called on the set.

Bloom Filters

- A specific implementation of a lossy set.
- $O(N)$ memory to store N keys with a fixed false-positive rate.
  - ... but with a very small constant (1 byte per key $\approx 1 - 2\%$ false positive rate).

Bloom Filters

Before

- Read file
- Find and return record for key

After

- If in-memory bloom filter returns false, return not-found
- Read file
- Find and return record for key

Tidy Data

Movies	Title	Lang	Runtime
	Princess Bride	English	98
	Princess Bride	Spanish	98
	Die Hard	English	132
	Die Hard	Polish	132
	The Matrix	English	136

Tidy Data

Movies	Title	Lang	Runtime
	Princess Bride	[English, Spanish]	98
	Die Hard	[English, Polish]	132
	The Matrix	English	136

Variable-length data is awkward to store and access

Tidy Data

Lang	Title	Lang
	Princess Bride	English
	Princess Bride	Spanish
	Die Hard	English
	Die Hard	Polish

Runtime	Title	Runtime
	Princess Bride	98
	Die Hard	132
	The Matrix	136

Functional Dependencies

Runtime	Title	Runtime
	Princess Bride	98
	Die Hard	132
	The Matrix	136

Each value for *Title* identifies a single value for *Runtime*.

Functional Dependency: $Title \rightarrow Runtime$

Functional Dependencies

R	A	B	C	D
	1	1	4	1
	1	2	4	2
	2	3	2	3
	3	5	1	4
	4	4	4	5

Functional Dependencies:

- $A \rightarrow C$
- $B \rightarrow A, C, D$
- $A, C \rightarrow D$
- $D \rightarrow A, B, C$

Keys

R	A	B	C	D
	1	1	4	1
	1	2	4	2
	2	3	2	3
	3	5	1	4
	4	4	4	5

- $B \rightarrow A, C, D$
- $D \rightarrow A, B, C$

B, D **are keys**
$\langle A, C \rangle$ **is a key**
$\langle A, B \rangle$ **is a super-key**

Normal Forms

- **Super-key**: A set of attributes with an FD to the rest of the table.
- **Key**: A minimal super-key.
- **1st Normal Form**: No Nesting, Lists, etc...
- **2nd Normal Form**: All FDs have a super-key on the left.

Recovering Denormalized Data

Lang	**Title**	**Lang**
	Princess Bride	English
	Princess Bride	Spanish
	Die Hard	English
	Die Hard	Polish

Runtime	**Title**	**Runtime**
	Princess Bride	98
	Die Hard	132
	The Matrix	136

We want to re-combine **Lang** and **Runtime**

Goal: Pair every element of **Lang** with the corresponding element of **Runtime**

Cartesian Product

R	A	B
	1	2
	2	1

S	B	C
	1	3
	1	4
	2	5

$R \times S$	A	R.B	S.B	C
	1	2	1	3
	1	2	1	4
	1	2	2	5
	2	1	1	3
	2	1	1	4
	2	1	2	5

Product + Filter

$Filter(R.B = S.B, \mathbf{R} \times \mathbf{S})$	**A**	**R.B**	**S.B**	**C**
	1	2	2	5
	2	1	1	3
	2	1	1	4

(This is called a Join)

(Since the filtering condition is an equality, it's an equi-join)

Equivalent SQL queries

```
SELECT *
FROM R, S
WHERE R.B = S.B
```

```
SELECT *
FROM R JOIN S ON R.B = S.B
```

```
SELECT *
FROM R NATURAL JOIN S
```

(Natural join = equi-join on all attributes with the same name)

Nested Loop Join

```
for r in R:
  for s in S:
    if condition(r, s):
      output <r, s>
```

Block-Nested Loop Join

```
for [r1 ... rB] in R:
  for s in S:
    for r in [r1 ... rB]
      if condition(r, s):
        output <r, s>
```

Sort-Merge Join

```
sort R
sort S
while Some(r) = R.peek() && Some(s) = S.peek():
  if r.key = s.key:
    # For simplicity, omitting case where multi
    output <r, s>
    r.next(); s.next()
  else if r.key < s.key:
    r.next()
  else;
    s.next()
```

In-Mem (1-Pass) Hash Join

```
hash_table = {}
for r in R;
  hash_table[r.key] += [r]
for s in S:
  for r in hash_table[s.key]:
    output <r, s>
```

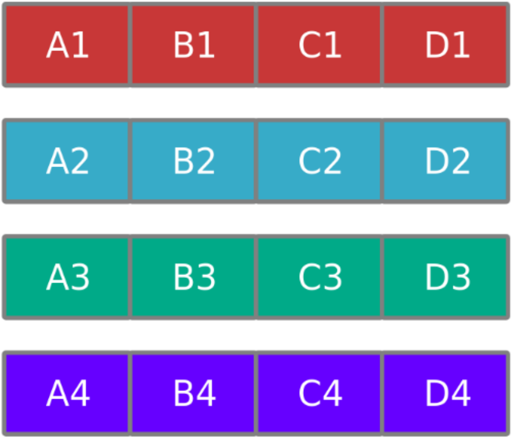
On-Disk (2-Pass) Hash Join

```
for i in 0 .. N:
  r_partitions[i] = new file
  s_partitions[i] = new file
for r in R:
  r_partitions[hash(r.key) mod i] += r
for s in S:
  s_partitions[hash(s.key) mod i] += s
for i in 0 .. n;
  in_mem_hash_join(r_partitions[i], s_partition
```

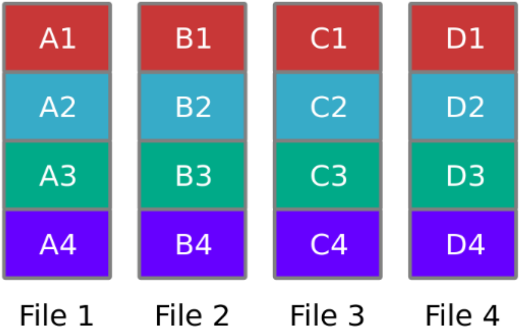
Join Summary

Alg.	Runtime	Mem	IO	Notes
NLJ	$O(\lvert R\rvert \times \lvert S\rvert)$	$O(1)$	$O(\lvert R\rvert \times \lvert S\rvert)$	
BNLJ	$O(\lvert R\rvert \times \lvert S\rvert)$	$O(B)$	$O\left(\frac{\lvert R\rvert \times \lvert S\rvert}{B}\right)$	B-size buffer
SMJ	$O(\lvert R\rvert + \lvert S\rvert + \lvert R \bowtie S\rvert) + O(sort)$	$O(1) + O(sort)$	$O(\lvert R\rvert + \lvert S\rvert) + O(sort)$	Equi-join only
1PHJ	Expected $O(\lvert R\rvert + \lvert S\rvert + \lvert R \bowtie S\rvert)$	$O(\lvert R\rvert + \lvert S\rvert)$	$O(\lvert R\rvert + \lvert S\rvert)$	Equi-join only
2PHJ	Expected $O(\lvert R\rvert + \lvert S\rvert + \lvert R \bowtie S\rvert)$	$O\left(\frac{\lvert R\rvert + \lvert S\rvert}{N}\right)$	$O(\lvert R\rvert + \lvert S\rvert)$	Equi-join only; N-size

Row Layouts



Columnar Layouts



Dictionary Encoding

People	Name	Country
r_1	Carl	Sweden
r_2	Ethan	United States Of America
r_3	Eric	United States Of America
r_4	Oliver	United Kingdom
r_5	Paul	United States Of America
r_6	Matt	United States Of America
r_7	Kelin	China

Country: Avg of 17 bytes per record.

Idea: Factorize!

Dictionary Encoding

People	Name	CountryID
r_1	Carl	1
r_2	Ethan	2
r_3	Eric	2
r_4	Oliver	3
r_5	Paul	2
r_6	Matt	2
r_7	Kelin	4

Countries	CountryID	Country
	1	Sweden
	2	United States Of America
	3	United Kingdom
	4	China

CountryID: 1 byte per record.

Country: +51 bytes.

Run-Length Encoding

People	Country
r_1	Sweden
r_2	United States Of America
r_3	United States Of America
r_4	United Kingdom
r_5	United States Of America
r_6	United States Of America
r_7	China

Country: Lots of redundancy

Run-Length Encoding

Idea 1: Run-Length encode!

People	Country
1 record	Sweden
2 records	United States Of America
1 record	United Kingdom
2 records	United States Of America
1 record	China

Idea 1: Replace each 'run' of the same value with (length, value) pairs.

Run-Length Encoding

People	Country
$\{r_1\}$	Sweden
$\{r_2, r_3, r_5, r_6\}$	United States Of America
$\{r_4\}$	United Kingdom
$\{r_7\}$	China

Idea 2: Group together IDs for each distinct value.

Data Model

Question: What is our atomic unit of data?

- One record?
- One page of data?
- One data table?

System-dependent; Let's talk about 'objects' instead

(An object can be a record, a range of records, a page of data, a data table, a partition, etc...)

Processes/Transactions

A process (i.e., transaction) is something that interacts with the data.

- A transaction can read from an object.
- A transaction can write to an object.

We model a process as a <u>sequence</u> of read and write operations.

We model several processes interacting with data as a sequence of read and write operations called **a schedule**.

Schedules

Given

- Objects A, B
- Process P1: R(A), R(B), W(A)
- Process P2: R(B), W(A)

The following is a schedule:

P1	**P2**
R(A)	
	R(B)
R(B)	
W(A)	
	W(A)

Serial Schedules

Question: What does it mean for a schedule to be correct?

Trivial Definition: Run all the processes, one at a time.
(no concurrency bugs at least)

P1	P2
R(A)	
R(B)	
W(A)	
	R(B)
	W(A)

P1	P2
	R(B)
	W(A)
R(A)	
R(B)	
W(A)	

We will assume that these are both correct: Both are **Serial** Schedules.

Some Interleaved Schedules are Correct

P1	P2
R(A)	
	R(B)
R(B)	
W(A)	
	W(A)

This schedule is indistinguishable, from the perspective of P1 or P2, from the serial schedule where P1 runs to completion first and then P2 runs to completion. It should be correct too.

A **Serializable** schedule is one where, given the available information, we can guarantee that there is no perceptible difference to processes accessing the data.

Problem: How do we formalize the idea of serializability in a general way?

Conflict Equivalence

Given two operations from different processes, we call the processes a **Conflict** iff:

- The operations act on the same object.
- At least one of the operations is a read.

Note that a conflict isn't necessarily bad... it's just a point where the two processes risk potential problems.

We call two schedules S_1 and S_2 **Conflict Equivalent** if we can reach S_2 by starting with S_1 and swapping the order of adjacent non-conflicting operations from different processes.

Conflict Serializable

A schedule is **conflict serializable** if it is conflict equivalent to some serial schedule.

Checking Conflict Serializability

Make a happens-before graph:

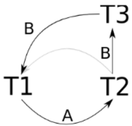
1. Create one node for each process
2. For each conflict, create an edge from the process of the earlier operation to the process of the later operation.

Time	T1	T2	T3
|	W(A)		
|		W(A)	
|		R(B)	
|			W(B)
↓	R(B)		

T1's write to A "happens before" T2's write

T2's read on B "happens before" T3's write

T3's write to B "happens before" T1's read



2-Phase Locking

Rules:

- A process must Lock an object before reading/writing it.
- A process must Unlock all of its held locks before it ends.
- Only one process may hold the lock on an object at a time.
- Once a process releases a lock, it may never again acquire a new lock (the 2-phase rule).

If processes follow the rules above, the resulting schedule will always be conflict serializable.
(However, some conflict serializable schedules can not be created by 2-phase locking.)

Reader/Writer Locks

Since Read/Read operations are not conflicts, we can relax the third rule and allow 2 processes to hold a lock concurrently if both are reading:

- If a process holds the writer lock for an object, no other process may hold a reader or writer lock on it.
- If a process holds a reader lock for an object, no other process may hold a writer lock on it.