Lecture 12: Indexing and IO Model

Announcements

1. Midterm

Notes

- [3.b]: 37%, [6.4] 56%
- Answer key posted

2. Project 2 → Project 3

(Training wheels in park)

(Explore the world. But tell us roughly where, by Friday)

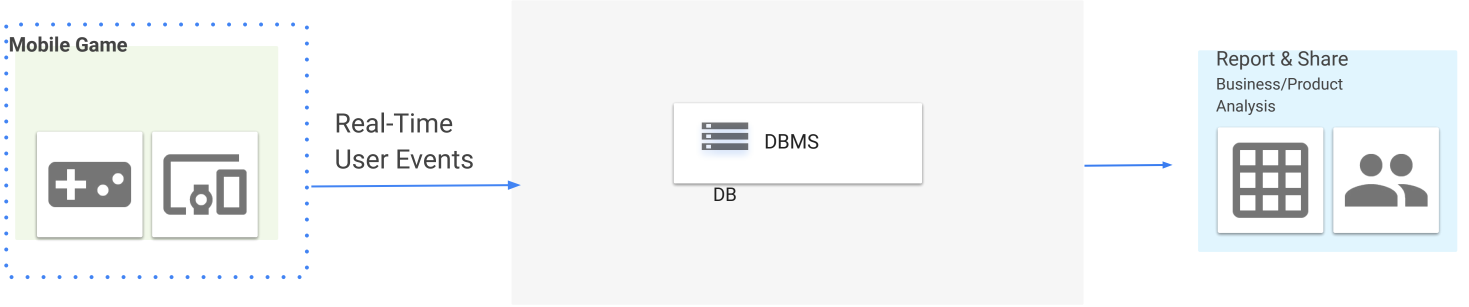

PAST
FUTURE

How?

Example Game App

DB v0

(Recap lectures)



Q1: 1000 users/sec writing?
Q2: Offline?
Q3: Support v1, v1’ versions?

Q7: How to model/evolve game data?
Q8: How to scale to millions of users?
Q9: When machines die, restore game state gracefully?

Q4: Which user cohorts?
Q5: Next features to build?
Experiments to run?
Q6: Predict ads demand?

App designer

Systems designer

Product/Biz designer

Today's Lecture

(Next weeks:
Scale, Scale, Scale)

1. Indexing

2. IO Model

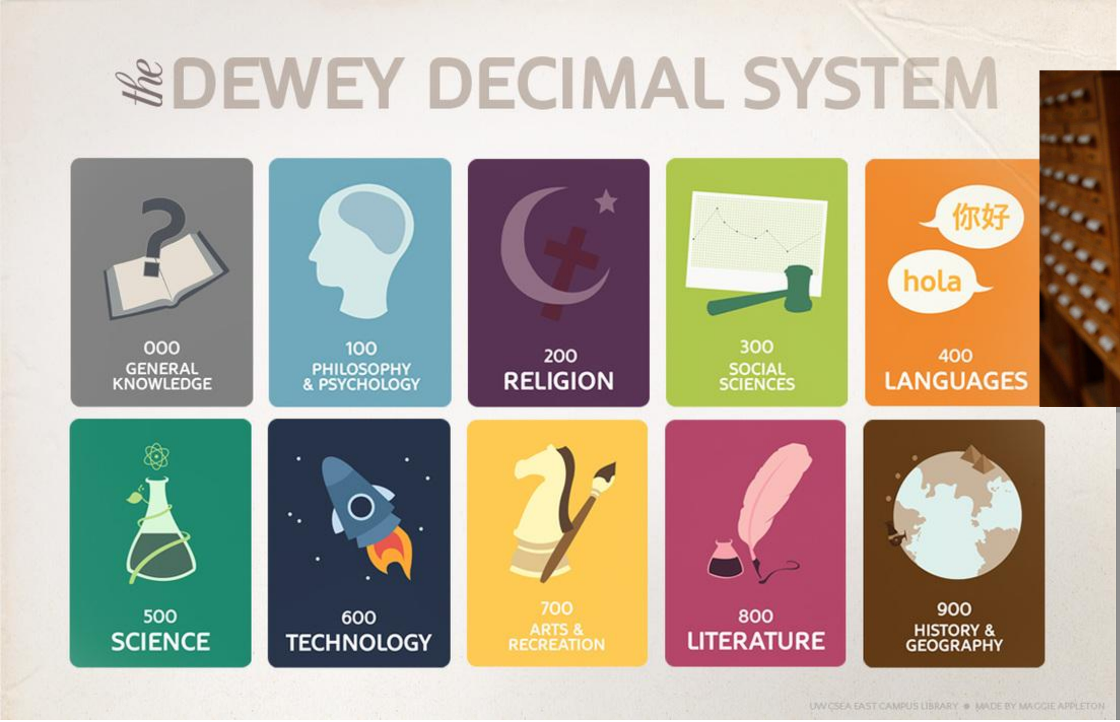
Example

Find Book in Library

Design choices?
- Scan through each aisle
- Lookup pointer to book location, with librarian's organizing scheme

Example

Find Book in Library With Index



[Index Cards](#)

Algorithm for book titles

- Find right category
- Lookup Index, find location
- Walk to aisle. Scan book titles. Faster if books are sorted

“If you don’t find it in the index, look very carefully through the entire catalog”

- Sears, Roebuck and Co., Consumers Guide, 1897

Latency numbers every engineer should know

Ballpark timings

execute typical instruction	1/1,000,000,000 sec = 1 nanosec
fetch from L1 cache memory	0.5 nanosec
fetch from L2 cache memory	7 nanosec
Mutex lock/unlock	25 nanosec
fetch from main memory	100 nanosec
send 2K bytes over 1Gbps network	20,000 nanosec
read 1MB sequentially from memory	250,000 nanosec
fetch from new disk location (seek)	8,000,000 nanosec
read 1MB sequentially from disk	20,000,000 nanosec
send packet US to Europe and back	150 milliseconds = 150,000,000 nanosec

(~0.25 msecs)

(~10 msecs)

(~20 msecs) (or use 100 MB/sec)

Example: Search for books

All books written by Rowling?’

Billion_Books

BID	Title	Author	Published	Full_text
	….	….	….	
7003	Harry Potter	Rowling	1999	...
1001	War and Peace	Tolstoy	1869	…
1002	Crime and Punishment	Dostoyevsky	1866	…
1003	Anna Karenina	Tolstoy	1877	…
	….			

```
SELECT *
FROM   Billion_Books
WHERE Author like ‘Rowling’
```

Example: Search for books

```
SELECT *
FROM   Billion_Books
WHERE Author like 'Rowling'
```

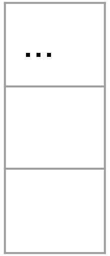
Input: Data size

1 Billion books

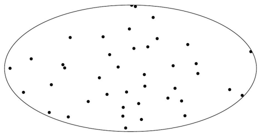
Each record = 1000 bytes

(i.e., 1000 GBs or 1 TB)

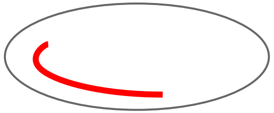
Design Choices



1. Data in RAM
   - Scan RAM sequentially & filter
     - Scan Time: 1000 GB * 0.25 msecs/1MB = 250 secs
     - Cost (@100$/16GB) ~= 6000$ of RAM



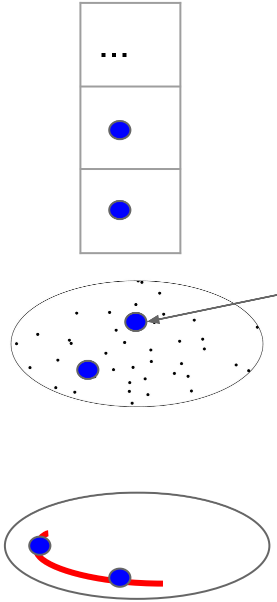
2. Data in disk (random spots)
   - Seek each record on disk & filter
     - Scan Time: (Seek) 10 msecs * 1Billion records + (Scan) 1 TB /100 MB-sec
       - = 10^7 secs (115 days) + 10^4 secs ~= 115 days
     - Cost (@100$/TB of disk) = 100$ of disk



3. Data in disk (sequentially organized)
   - Seek to table, and sequentially scan records on disk & filter
     - Scan Time: (Seek) 10 msecs + (Scan) 1 TB /100 MB-sec
       - = 10^4 secs ~= 3 hrs
     - Cost (@100$/TB of disk) = 100$ of disk

Example: Search for books

```
SELECT *
FROM   Billion_Books
WHERE Author like ‘Rowling’
```



<Disk block, position>

Index in RAM

Location	Author
	Rowling
	Tolstoy
	Rowling
	...

Index => Maintain location of record

- Memory block
- Disk block (seek positions)

Notes:

- O(n) seeks for ‘n’ results
- RAM index costs $$ but speedsup
- Or index on disk (later)
- Or index on index on index….(later)

Indexes on a table

- An index speeds up selections on search key (s)
  - Any subset of fields
- Example

Books(BID, name, author, price, year, text)

On which attributes would you build indexes?

Example

Billion_Books

BID	Title	Author	Published	Full_text
1001	*War and Peace*	Tolstoy	1869	…
1002	*Crime and Punishment*	Dostoyevsky	1866	…
1003	*Anna Karenina*	Tolstoy	1877	…

```
SELECT *
FROM   Billion_Books
WHERE Published > 1867
```

Example

By_Yr_Index

Published	BID
1866	1002
1869	***1001***
1877	***1003***
...	

Billion_Books

BID	Title	Author	Published	Full_text
1001	*War and Peace*	Tolstoy	1869	…
1002	*Crime and Punishment*	Dostoyevsky	1866	…
1003	*Anna Karenina*	Tolstoy	1877	…
...				

Maintain an index for this, and search over that!

Why might just keeping the table sorted by year not be good enough?

Example

By_Yr_Index

Published	BID
1866	1002
1869	1001
1877	1003

By_Author_Title_Index

Author	Title	BID
Dostoyevsky	Crime and Punishment	1002
Tolstoy	Anna Karenina	1003
Tolstoy	War and Peace	1001

Russian_Novels

BID	Title	Author	Published	Full_text
1001	*War and Peace*	Tolstoy	1869	…
1002	*Crime and Punishment*	Dostoyevsky	1866	…
1003	*Anna Karenina*	Tolstoy	1877	…

Can have multiple indexes to support multiple search keys

Indexes shown here as tables, but in reality we will use more efficient data structures…

Covering Indexes

By_Yr_Index

Published	BID
1866	1002
1869	1001
1877	1003

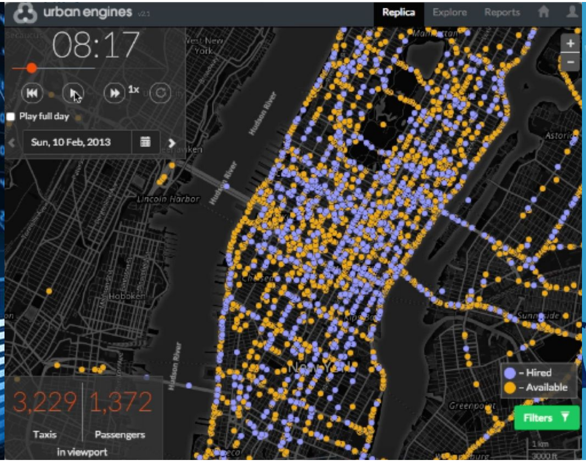
An index **<u>covers</u>** *for a specific query* if the index contains all the needed attributes- ***meaning the query can be answered using the index alone!***

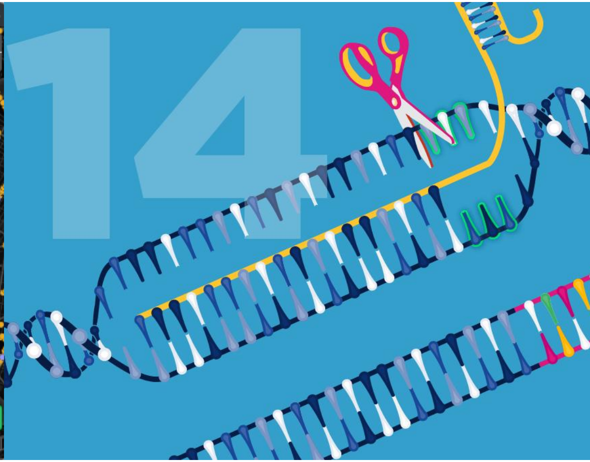
The “needed” attributes are the union of those in the SELECT and WHERE clauses…

Example:

```
SELECT Published, BID
FROM   Billion_Books
WHERE Published > 1867
```

Kinds of Indexes (different data types)





Index for Strings, Integers

Time series, GPS traces, Genomes, Video sequences

Advanced: Equality vs Similarity,
Ranges, Subsequences
Composites of above

Indexes (definition)

An *index* is a **data structure** mapping search keys to sets of rows in table

- Provides efficient lookup & retrieval by search key value (usually much faster than scanning all rows and searching)

An index can store

- full rows it points to (*primary index*), OR
- pointers to rows (*secondary index*) [much of our focus]

Operations on an Index

- <u>Search</u>: Quickly find all records which meet some *condition on the search key attributes*
  - (Advanced: across rows, across tables)

- <u>Insert / Remove</u> entries
  - Bulk Load / Delete. Why?

Indexing is one the most important features provided by a database for performance

Big Scaling (with Indexes)

Roadmap

Hashing

Sorting

Counting

Hashing-Sorting-Counting solves (virtually) “all” known problems :=)

+ Boost with a few patterns -- Cache, Parallelize, Pre-fetch

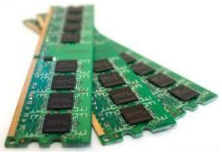
THE BIG IDEA

Note
Works for Relational, noSQL, OKVs
(e.g. mySQL, postgres, BigQuery, BigTable, MapReduce, Spark)

Big Scaling (with Indexes)

Roadmap

Primary data structures/algorithms



Hashing	Sorting	Counting
HashTables ($hash_i$(key) --> value)	BucketSort, QuickSort MergeSort	HashTable + Counter ($hash_i$(key) --> <count>)

?????

1. The Buffer

Transition to Indexing Mechanisms

1. So you can **understand** what the database is doing!
   - Understand the CS challenges of a database and how to use it.
   - Understand how to optimize a query

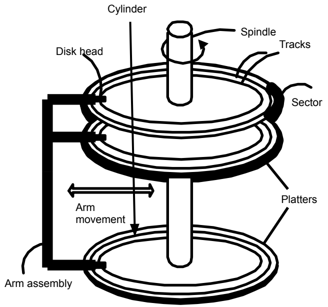
2. Many **mechanisms** have become **stand-alone systems**
   - **Indexing** to Key-value stores
   - Embedded join processing
   - SQL-like languages take some aspect of what we discuss (PIG, Hive)

What you will learn about in this section

1. RECAP: Storage and memory model

2. Buffer primer

High-level: Disk vs. Main Memory



<u>**Disk**</u>:

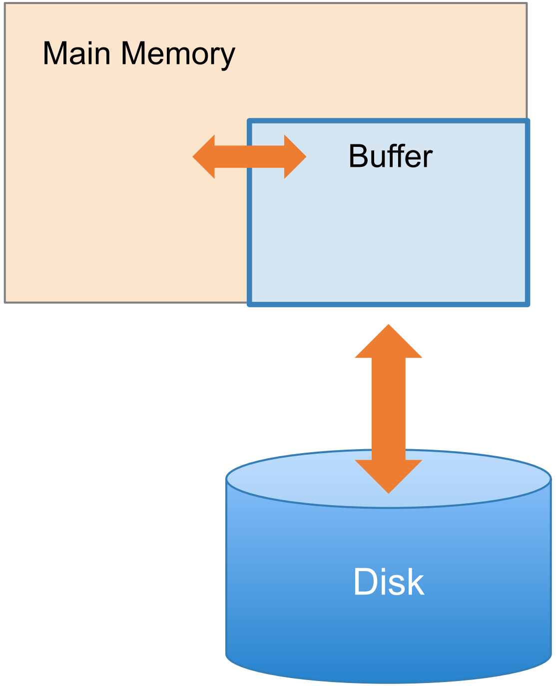
- ***Slow:*** Sequential *block* access
  - Read a blocks (not byte) at a time, so sequential access is cheaper than random
  - **Disk read / writes are expensive!**
- ***Durable:*** We will assume that once on disk, data is safe!
- ***Cheap***

<u>**Random Access Memory (RAM) or Main Memory:**</u>

- ***Fast:*** Random access, byte addressable
  - ~10x faster for <u>sequential access</u>
  - ~100,000x faster for <u>random access!</u>
- ***Volatile:*** Data can be lost if e.g. crash occurs, power goes out, etc!
- ***Expensive:*** For $100, get 16GB of RAM vs. 2TB of disk!

The Buffer

- A **buffer** is a part of physical memory used to store *temporary data*
    - *In this lecture:* a region in main memory used to store **intermediate data between disk and processes**
- Why? Reading / writing to disk is slow-need to cache data!

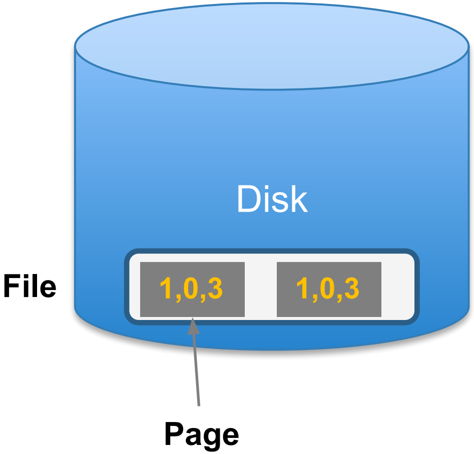


A Simplified Filesystem Model

- For us, a **page** is ***fixed-sized array*** of memory
  - Think: One or more disk blocks
  - Interface: write to an entry (called a **slot**) or set to “None”

  - DBMS also needs to handle variable length fields
    - Page layout is key for good hardware utilization (in cs 346)

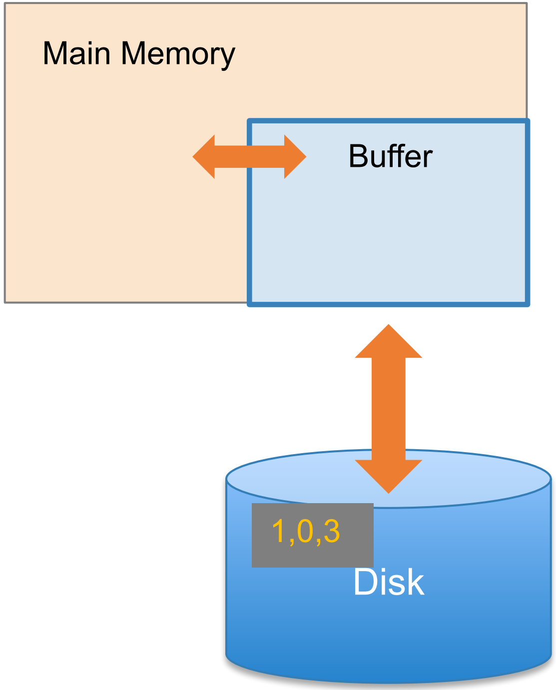
- And a **file** is a *variable-length list* of pages
  - Interface: create / open / close; next_page(); etc.



The (Simplified) Buffer

Buffer located in **main memory** operates over **pages** and **files**:

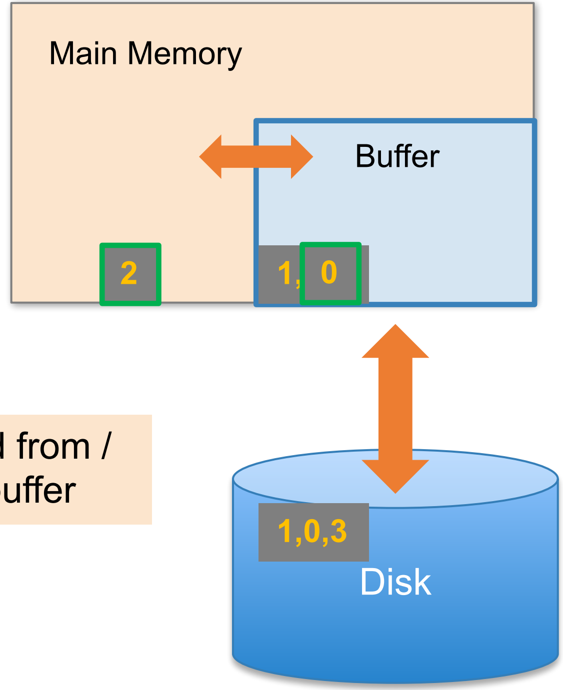
- <u>**Read(page):**</u> Read page from disk --> buffer *if not already in buffer*



The (Simplified) Buffer

Buffer located in **main memory** operates over **pages** and **files**:

- <u>**Read(page):**</u> Read page from disk --> buffer *if not already in buffer*

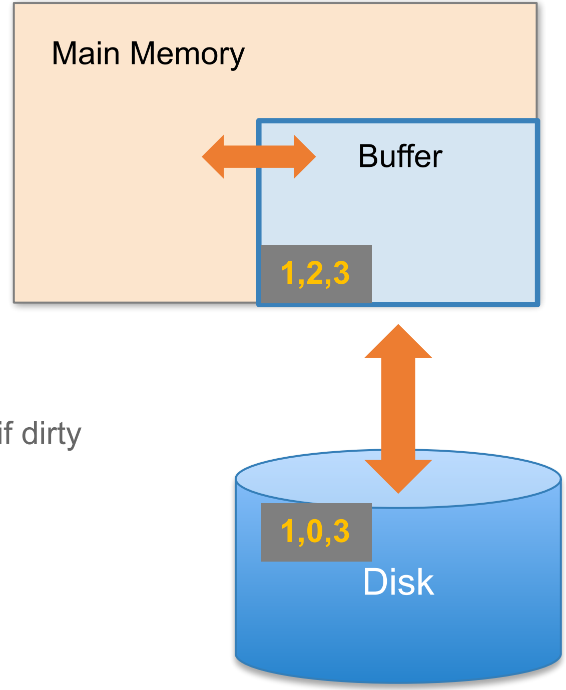


Processes can then read from / write to the page in the buffer

The (Simplified) Buffer

Buffer located in **main memory** operates over **pages** and **files**:

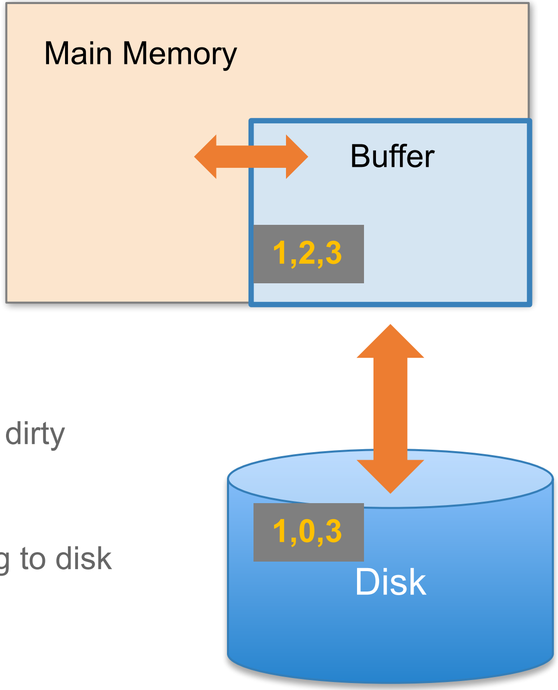
- **<u>Read(page):</u>** Read page from disk --> buffer *if not already in buffer*
- **<u>Flush(page):</u>** Evict page from buffer & write to disk, if dirty (dirty ⇒ modified page)



The (Simplified) Buffer

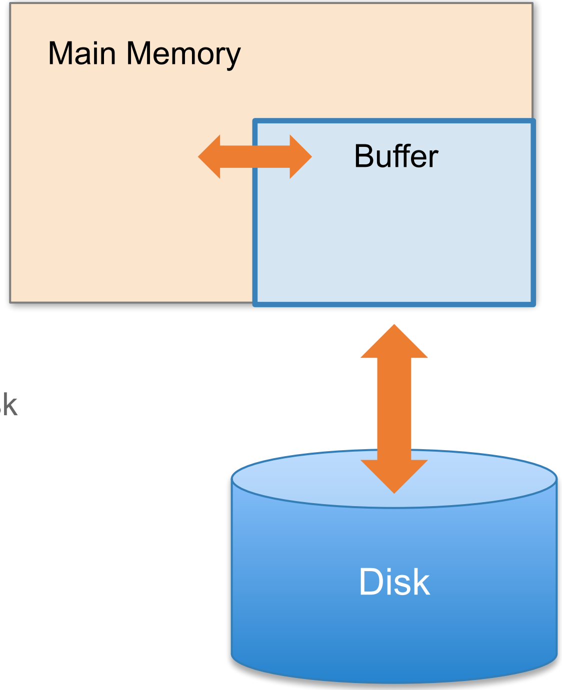
Buffer located in **main memory** operates over **pages** and **files**:

- <u>**Read(page):**</u> Read page from disk --> buffer *if not already in buffer*
- <u>**Flush(page):**</u> Evict page from buffer & write to disk if dirty
- <u>**Release(page):**</u> Evict page from buffer *without* writing to disk



Managing Disk: The DBMS Buffer

- Database maintains its own buffer
  - Why? The OS already does this…
  - DB knows more about access patterns
  - Recovery and logging require ability to **flush** to disk



The Buffer Manager

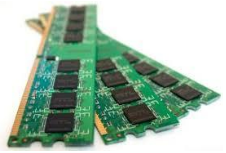
- A **buffer manager** manages operations for the buffer:
  - Primarily, handles & executes the "replacement policy"
    - i.e. finds a page in buffer to flush/release if buffer is full and a new page needs to be read in
  - DBMSs typically implement their own buffer management routines

2. External Merge Algorithm

Big Scaling (with Indexes)

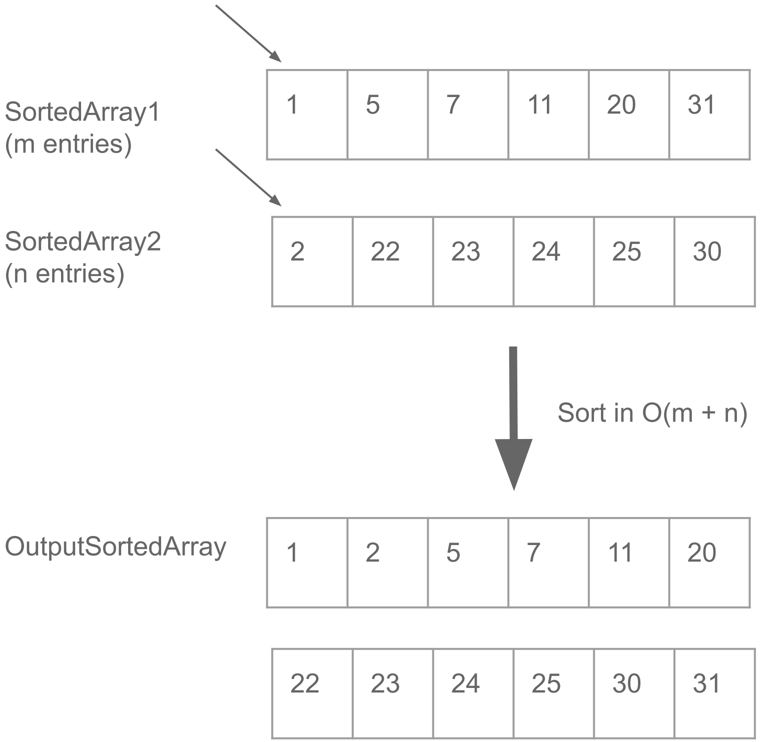
Roadmap

Primary data structures/algorithms

	Hashing	Sorting	Counting
	HashTables ($hash_i$(key) --> value)	BucketSort, QuickSort MergeSort	HashTable + Counter ($hash_i$(key) --> <count>)
		MergeSortedFiles, MergeSort	
			

MergeSortedFiles

(in RAM)



Challenge: Merging Big Files with Small Memory

How do we *efficiently* merge two sorted files when both are much larger than our main memory buffer?

Key point: Disk IO (R/W) dominates the algorithm cost

Our first example of an **“IO aware”** algorithm / cost model

External Merge Algorithm

- **Input**: 2 sorted lists of length M and N
- **Output:** 1 sorted list of length M + N
- **Required:** At least 3 Buffer Pages
- **IOs**: 2(M+N)

Key (Simple) Idea

To find an element that is no larger than all elements in two lists, one only needs to compare minimum elements from each list.

If:

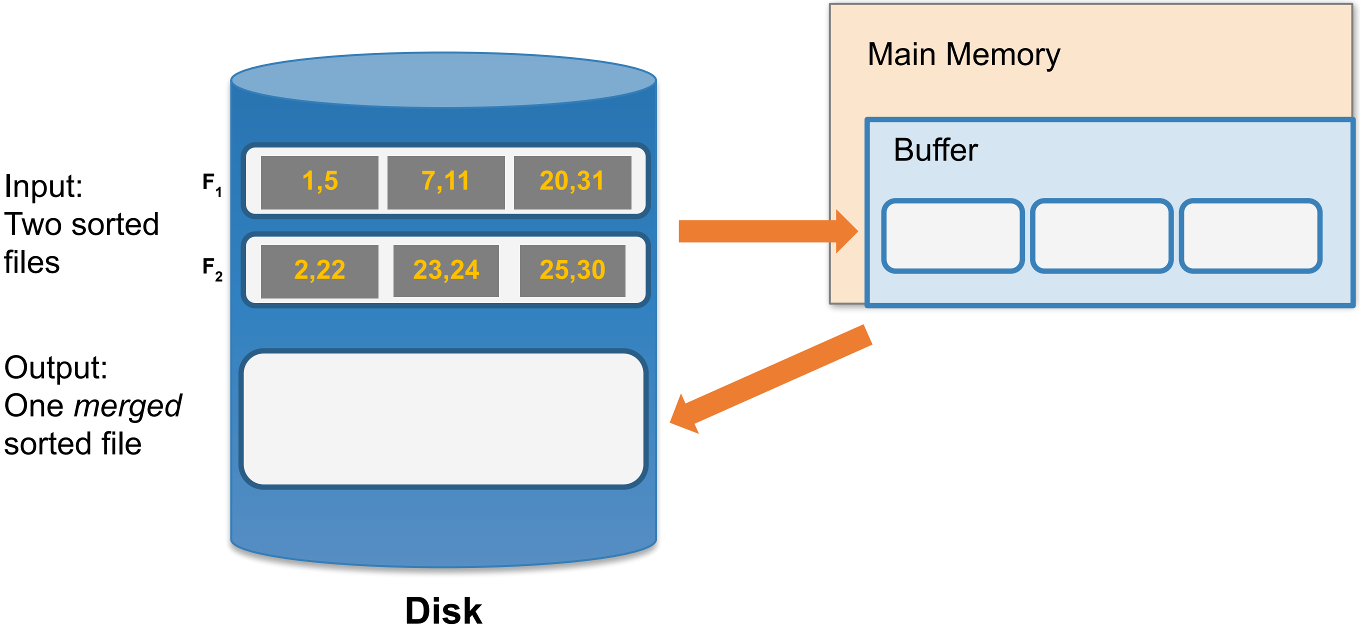
$$A_1 \leq A_2 \leq \cdots \leq A_N$$
$$B_1 \leq B_2 \leq \cdots \leq B_M$$

Then:

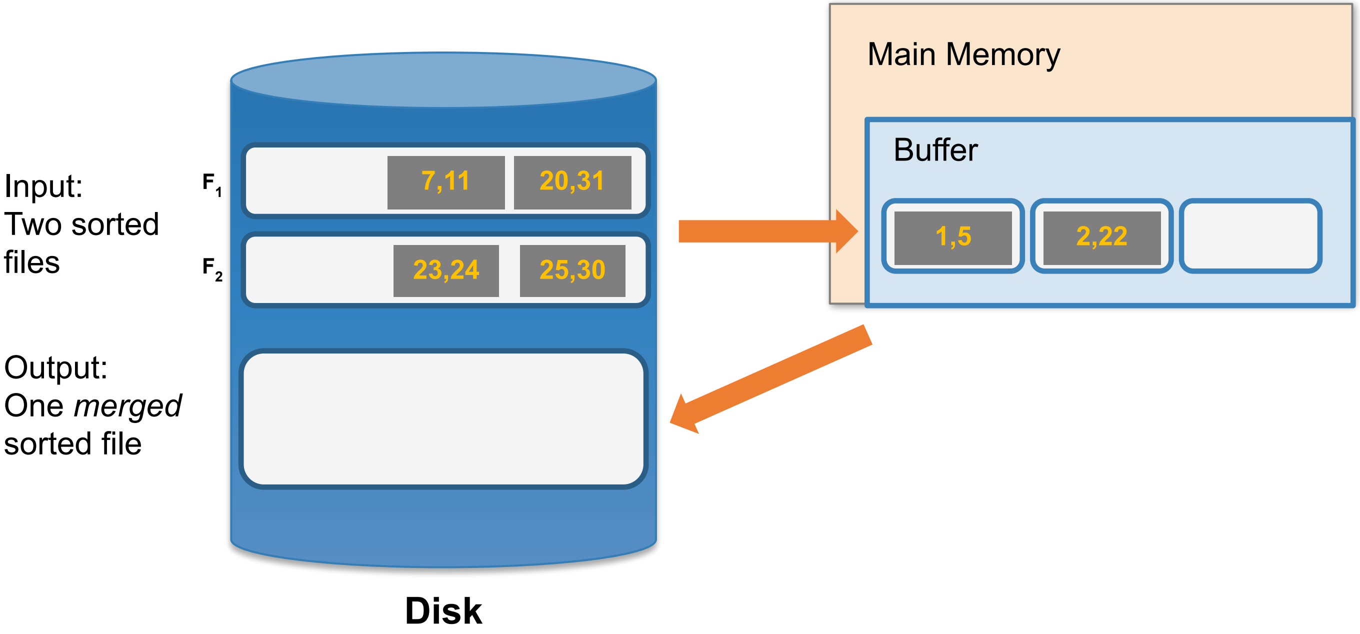
$$Min(A_1, B_1) \leq A_i$$
$$Min(A_1, B_1) \leq B_j$$

for i=1….N and j=1….M

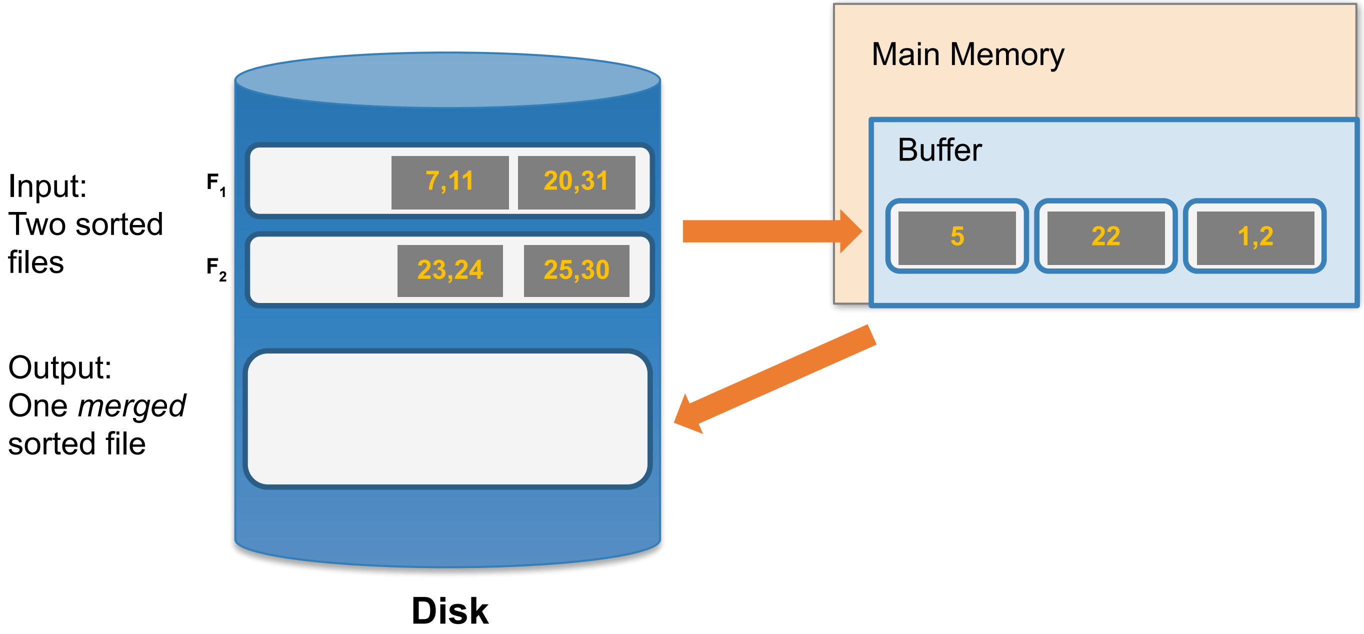
External Merge Algorithm



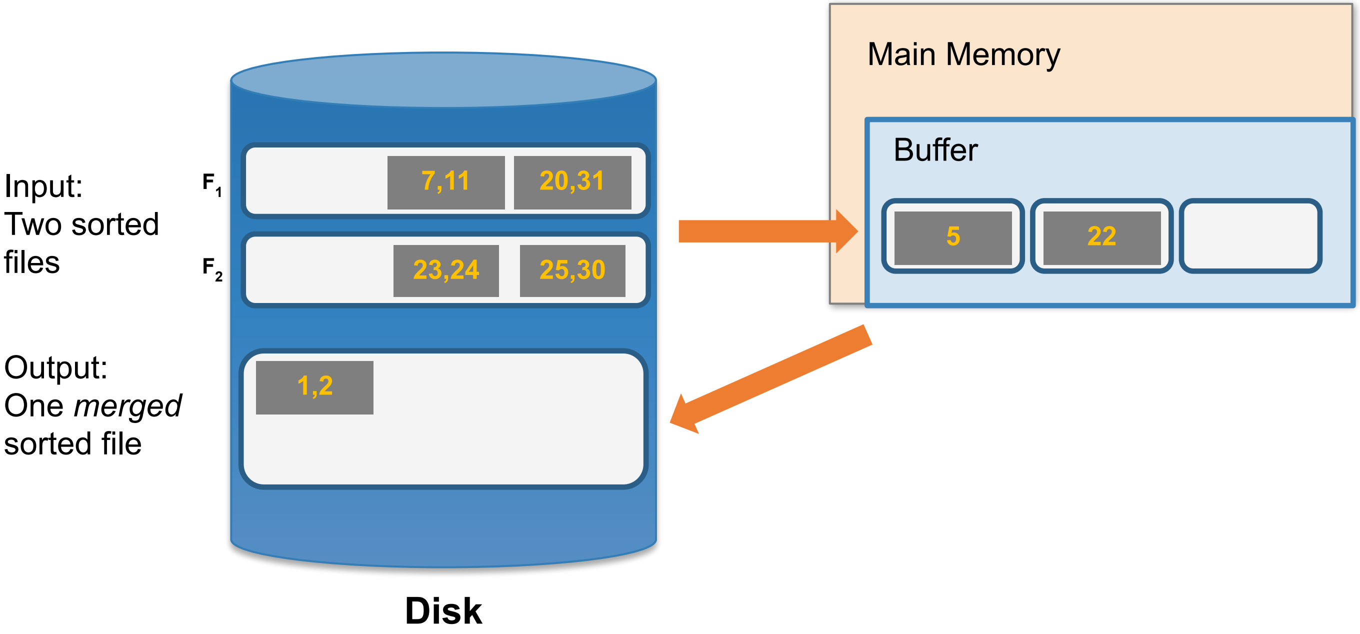
External Merge Algorithm



External Merge Algorithm



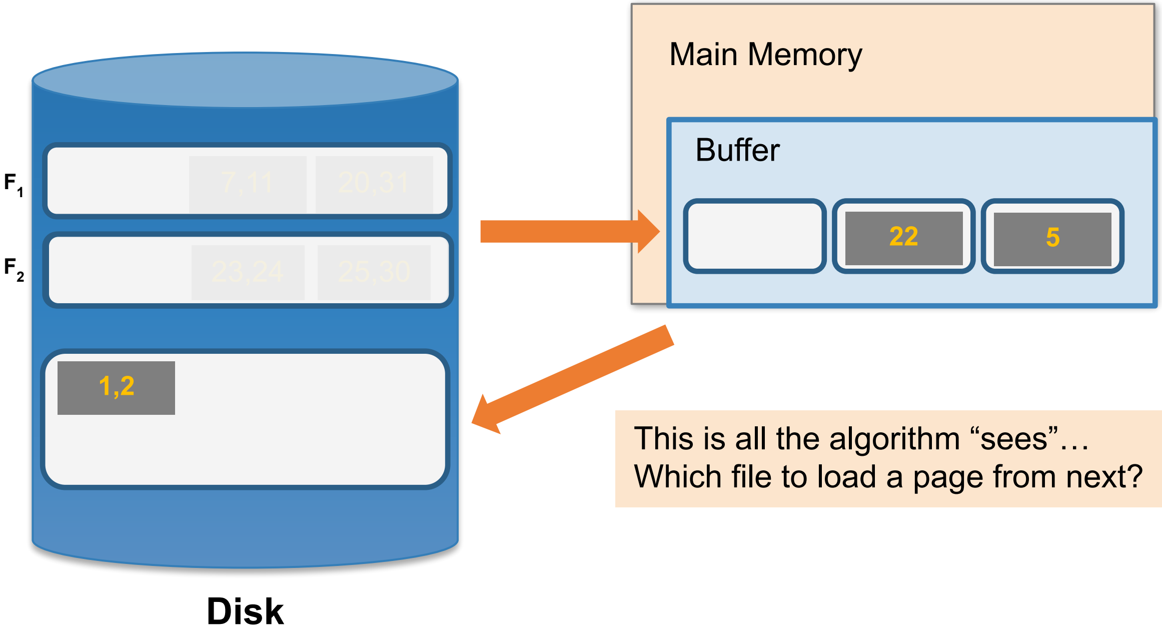
External Merge Algorithm



External Merge Algorithm

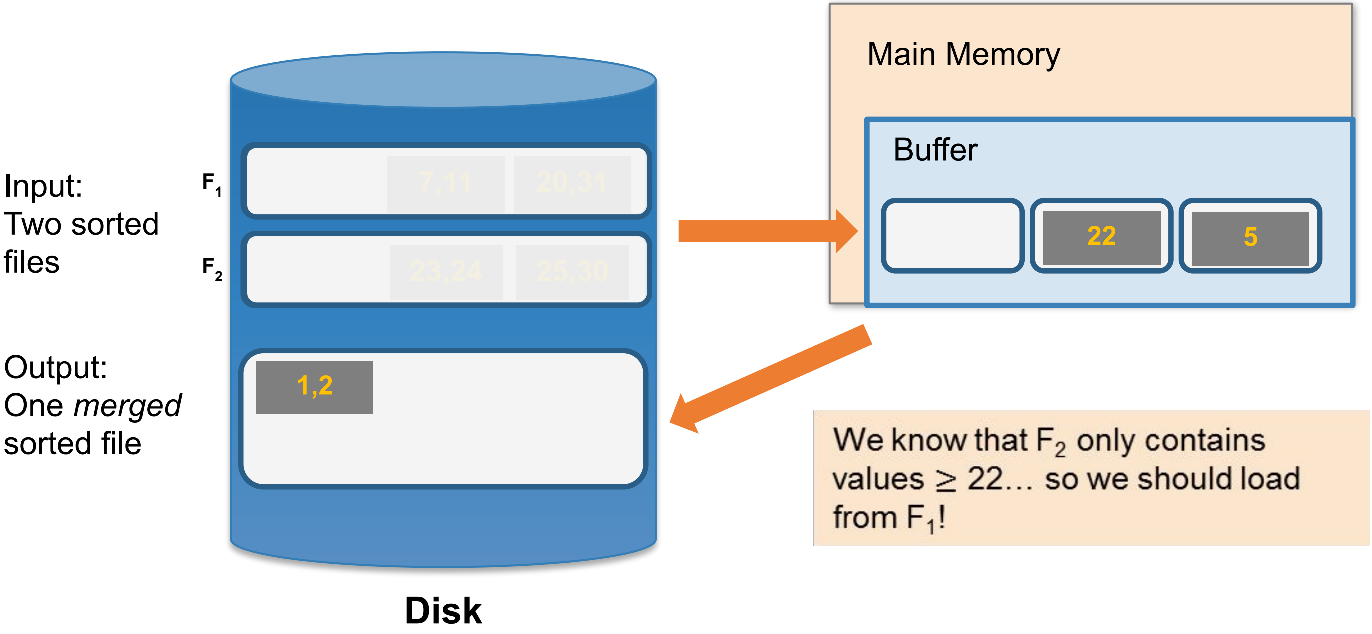
Input:
Two sorted files

Output:
One *merged* sorted file



This is all the algorithm “sees”…
Which file to load a page from next?

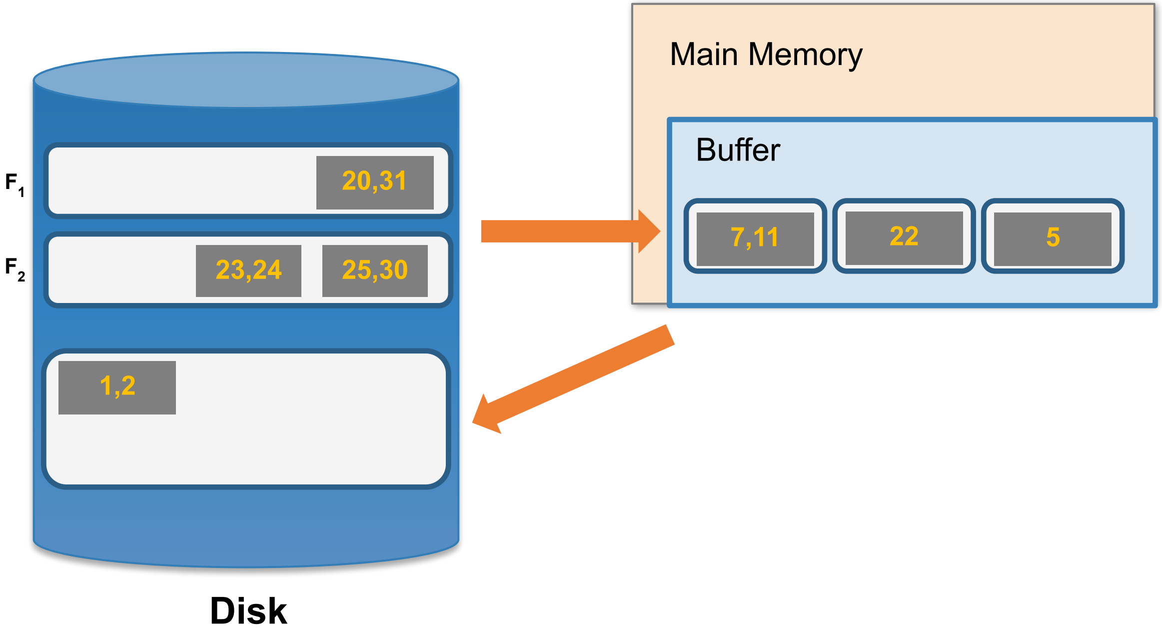
External Merge Algorithm



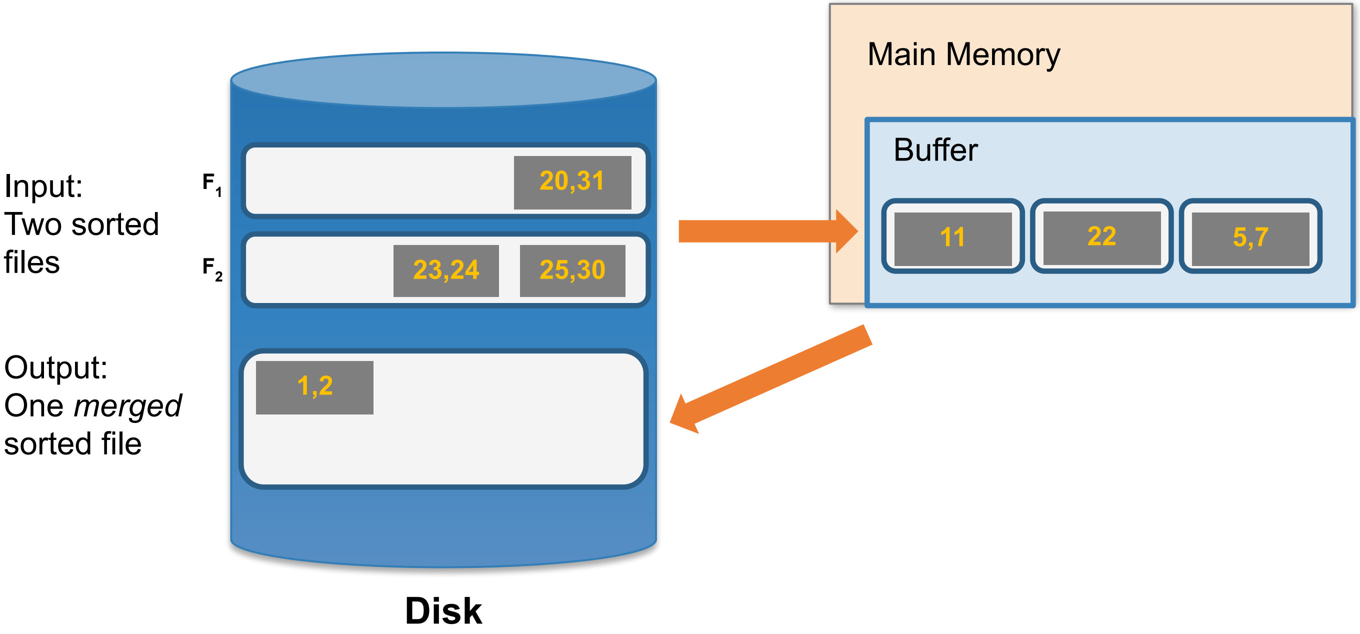
External Merge Algorithm

Input:
Two sorted files

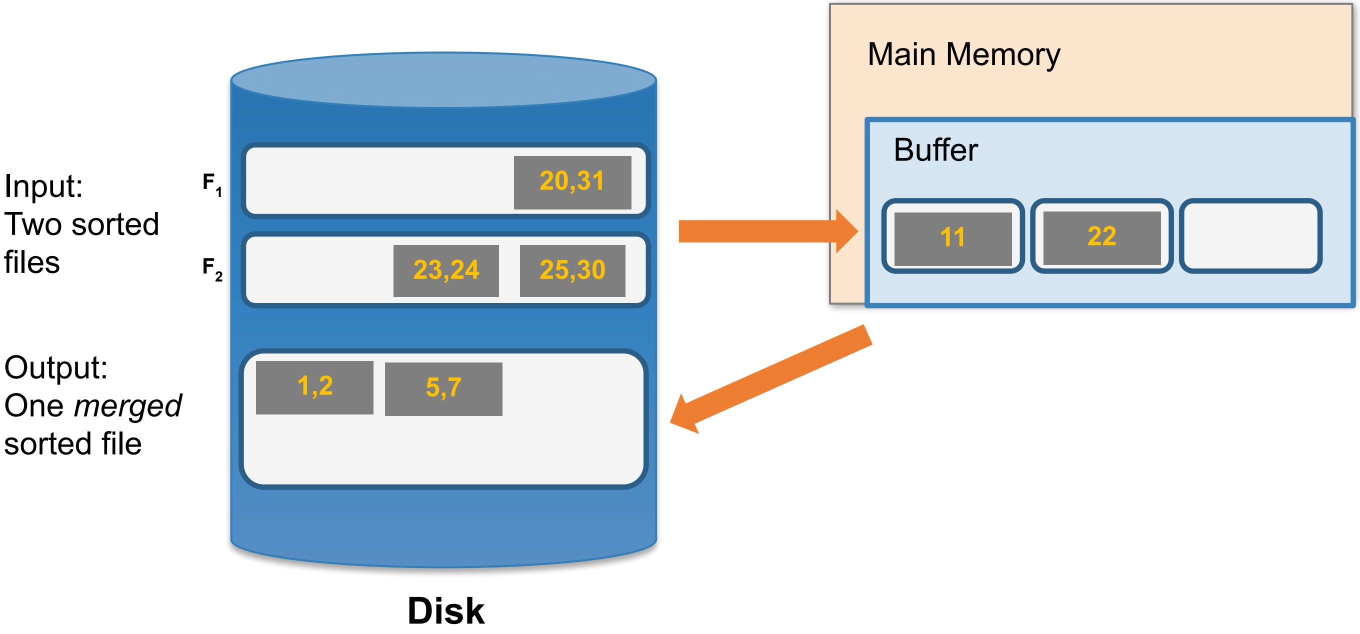
Output:
One *merged* sorted file



External Merge Algorithm



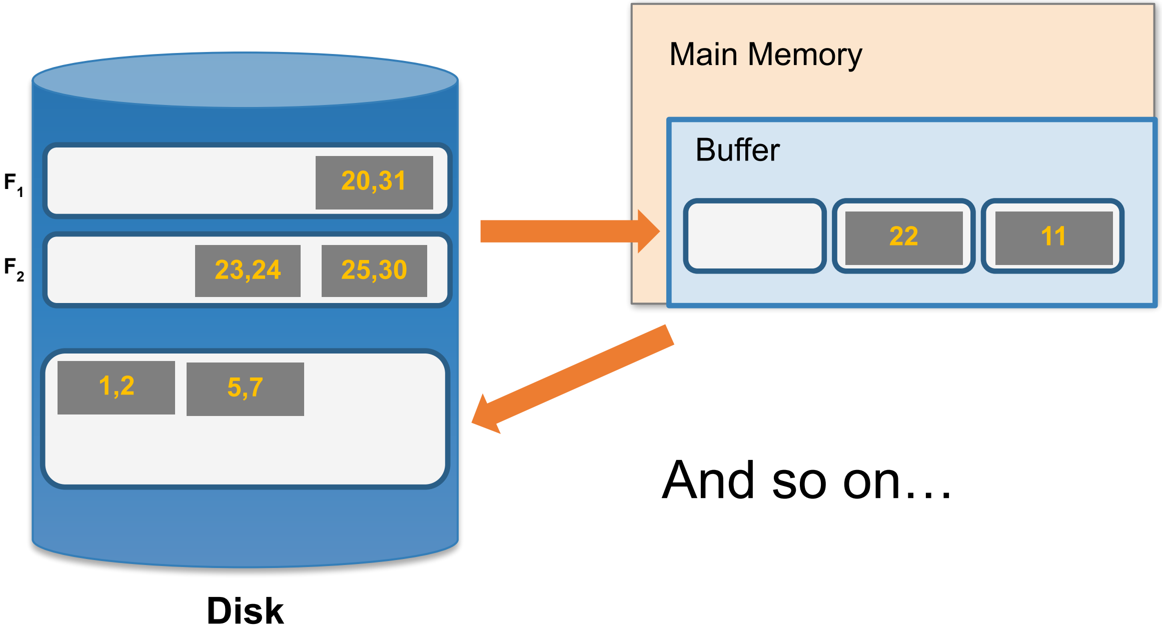
External Merge Algorithm



External Merge Algorithm

Input:
Two sorted files

Output:
One *merged* sorted file



And so on…

We can merge lists of arbitrary length with only 3 buffer pages.

If lists of size M and N, then
Cost: 2(M+N) IOs
Each page is read once, written once

With B+1 buffer pages, can merge B lists. How?

Recap: External Merge Algorithm

- Suppose we want to merge two **sorted** files both much larger than main memory (i.e. the buffer)

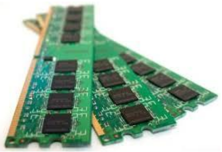
- We can use the **external merge algorithm** to merge files of ***arbitrary length*** in **2*(N+M) IO** operations with only **3 buffer pages**!

Our first example of an "IO aware" algorithm / cost model

Big Scaling (with Indexes)

Roadmap

Primary data structures/algorithms



Hashing	Sorting	Counting
HashTables ($hash_i$(key) --> value)	BucketSort, QuickSort MergeSort	HashTable + Counter ($hash_i$(key) --> <count>)
	MergeSortedFiles ?????	