1

Announcements

Project 1 due Friday

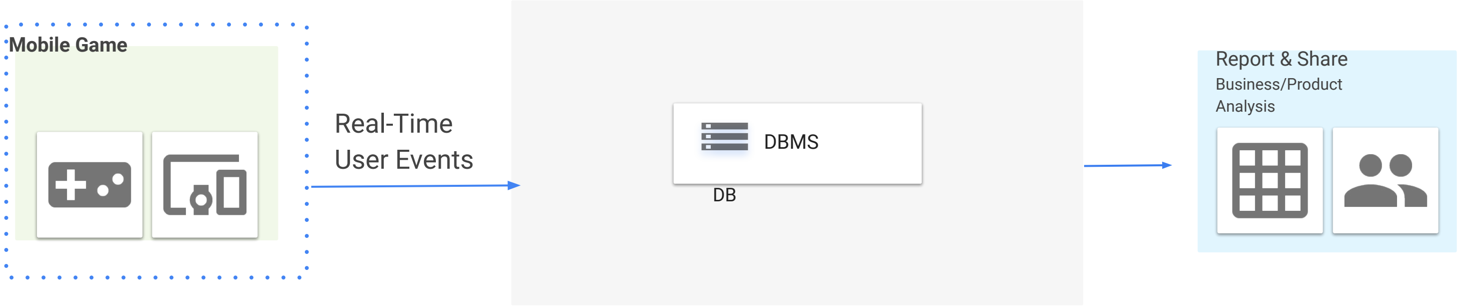
2

How?

Example Game App

DB v0

(Recap lectures)



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

App designer

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

Systems designer

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

Product/Biz designer

Scale:
Logical → Physical DB

4

Logical → Physical?

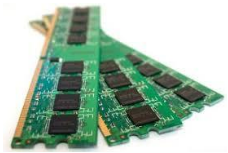
Company(<u>CName</u>, StockPrice, Date, Country)

Logical Table

			Col3	
	Company			
	CName	Date	Price	Country
Row1	AAPL	Oct1	101.23	USA
	AAPL	Oct2	102.25	USA
Row3	AAPL	Oct3	101.6	USA
	GOOG	Oct1	201.8	USA
Row5	GOOG	Oct2	201.61	USA
	GOOG	Oct3	202.13	USA
	Alibaba	Oct1	407.45	China
Row8	Alibaba	Oct2	400.23	China

So far... how to run SQL on “logical tables?”

Next: How to store table in physical storage ‘files’?
How to access rows/columns?
(e.g., disk, RAM, clusters)

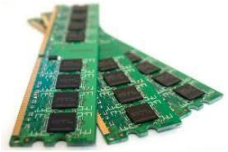


5

Logical → Physical?

Small tables (e.g., < 1 GB)? “Easy” with tools you already know

- Data structures? Linked lists, arrays, trees, hash tables
- Algorithms? Sorting, Hashing, Dynamic Programming, Graph algorithms, etc.
- I/O model ⇒ Work with data in RAM



Big tables? (e.g., TBs, PetaBytes?)

⇒ “Easy” with tools from cs145 + advanced systems classes

6

In this Section

(Next weeks:
Scale, Scale, Scale)

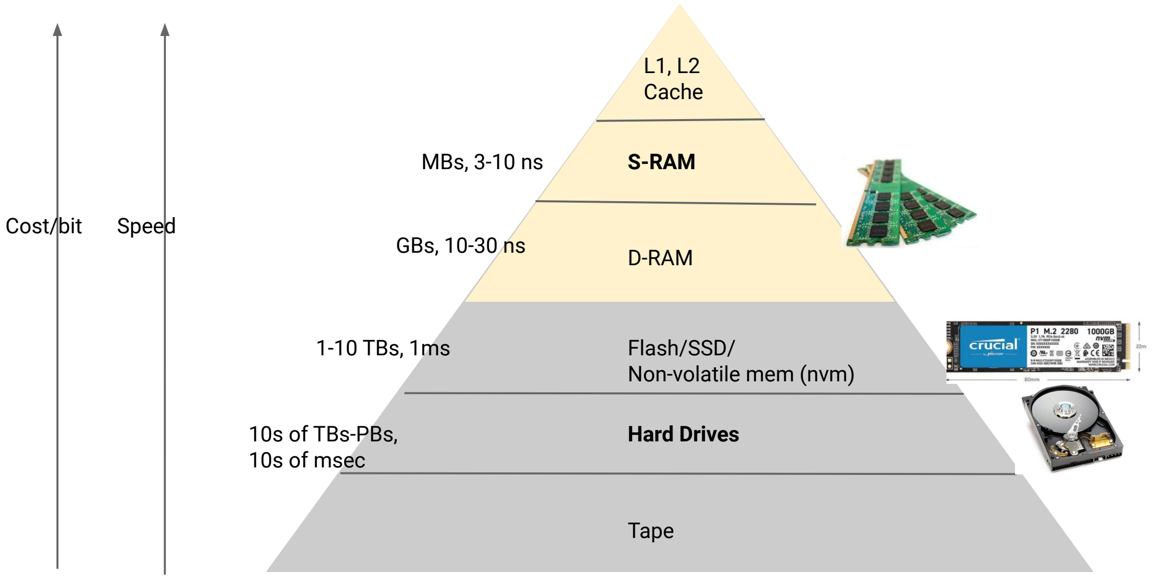
1. IO Model
2. Data layout
3. Indexing
4. Organizing Data and Indices

7

IO Hierarchy

Volatile -- data lost when you turn off power

Non-Volatile



Rough rule of thumb

<1-10 GBs
Usual CS algorithms
Pandas + SQL

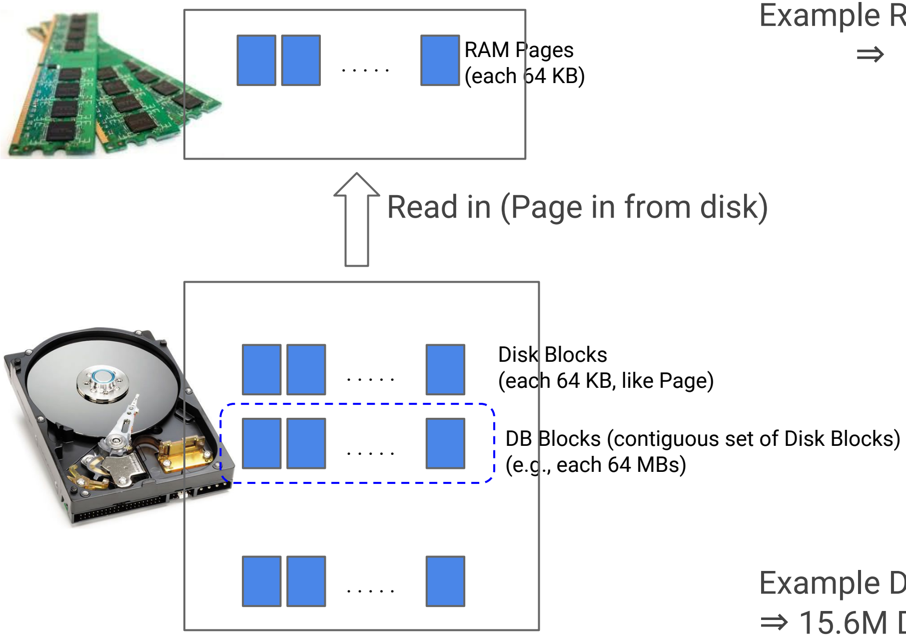
> 10 GBs - 10 TBs
SQL on a cluster
Store on SSDs

> 10 TBs - PBs
SQL + Cs145 algorithms
Store on HDs

⇒ Rest of cs145: Focus on simplified RAM + Disk model
(learn tools for other IO models)

8

Basic IO Model for Reads



Example RAM size: 64 GB RAM
⇒ 1 million 64 KB pages

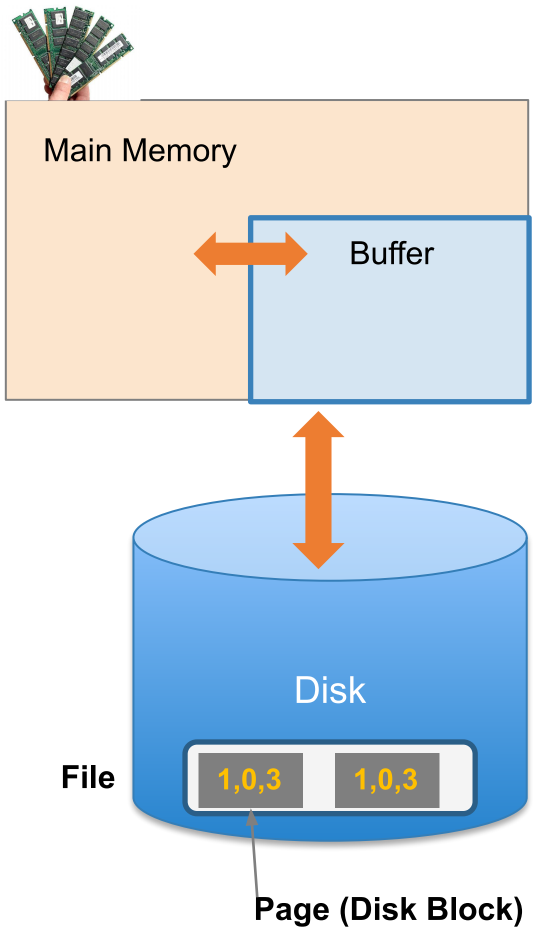
Example Disk size: 1 TB
⇒ 15.6M Disk Blocks (= 1 TB/64 KB)

⇒ 15.6K DB Blocks (@ 64 MB/DB block)

In DBs, Page and Disk Block are usually same size.
⇒ In this class, we'll use them interchangeably

DB Buffer in RAM

A buffer is a part of physical memory used to store intermediate data between disk and processes



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
- Buffer Manager handles page replacement policies

10

In this Section

(Next weeks:
Scale, Scale, Scale)

1. IO Model
2. Data layout
3. Indexing
4. Organizing Data and Indices

11

Data Layout

Company(CName, StockPrice, Date, Country)

Logical Table

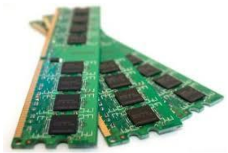
How to store table in physical storage 'files'? (e.g., disk, RAM)

Col3

	Company			
	CName	Date	Price	Country
Row1	AAPL	Oct1	101.23	USA
	AAPL	Oct2	102.25	USA
Row3	AAPL	Oct3	101.6	USA
	GOOG	Oct1	201.8	USA
Row5	GOOG	Oct2	201.61	USA
	GOOG	Oct3	202.13	USA
	Alibaba	Oct1	407.45	China
Row8	Alibaba	Oct2	400.23	China

...

Row Billion



12

Data Layout

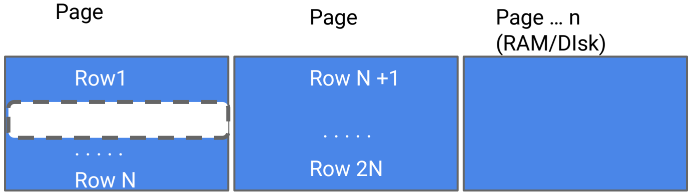
Company(CName, StockPrice, Date, Country)

Logical Table

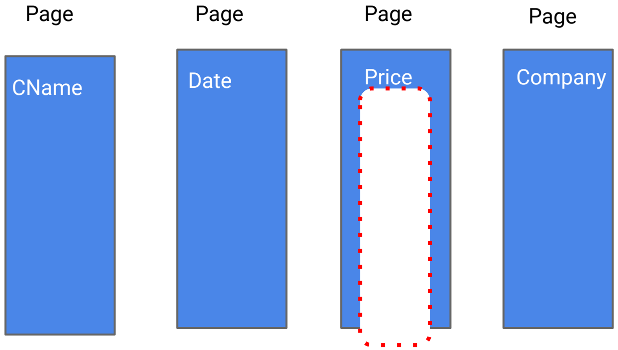
Col3

	Company			
	CName	Date	Price	Country
Row1	AAPL	Oct1	101.23	USA
	AAPL	Oct2	102.25	USA
Row3	AAPL	Oct3	101.6	USA
	GOOG	Oct1	201.8	USA
Row5	GOOG	Oct2	201.61	USA
	GOOG	Oct3	202.13	USA
	Alibaba	Oct1	407.45	China
Row8	Alibaba	Oct2	400.23	China

Row Billion



Row based storage (aka Row Store)



Column based storage (aka Column Store)

13

Data Layout

Company(<u>CName</u>, StockPrice, Date, Country)

Col3

	Company			
	CName	Date	Price	Country
Row1	AAPL	Oct1	101.23	USA
	AAPL	Oct2	102.25	USA
Row3	AAPL	Oct3	101.6	USA
	GOOG	Oct1	201.8	USA
Row5	GOOG	Oct2	201.61	USA
	GOOG	Oct3	202.13	USA
	Alibaba	Oct1	407.45	China
Row8	Alibaba	Oct2	400.23	China

Row based storage
(aka Row Store)

- Easy to retrieve and modify full tuple/row
- Classic way to organize data

Column based storage
(aka Column Store)

- Aggregation queries -- e.g., AVG(Price)
- Compression -- e.g., see Date column
- Scale to machine clusters -- distribute columns to different machines
- Only retrieve columns you need for query
- Cons: Updates are more work

Tradeoffs on ‘Workloads:’

1. Analytics: Lots of data, many exporatoratory queries on few columns, e.g., Youtube analytics
2. Transactions: Good combination of reads and writes, e.g., Delta Airlines

14

Example

Origin Story of BigQuery (Dremel)

WebPage(URL, PageRank, Language, NumVisits, HTML)

Google index of Web Pages (~2005)

URL: 100 bytes
PageRank: 8 bytes
Language: 4 bytes
Number of visitors: 4 bytes
HTML: 2 MBs * 5 versions ← (the big column)

⇒ Overall size = ~10 MBs/URL, stored in row format

Use case: What's PageRank of popular pages?

- E.g., select AVG(PageRank) … where NumVisits > 100
- **Hours** to run query over 1 billion URLs. Why?
  ⇒ Row based layout: Processing 10 MB*1 billion urls
  ⇒ Column based layout: Need to process only 12 bytes * 1 billion urls (1 million times faster)
- **Core idea:** Exploratory queries usually focus on a few columns

15

Example

Origin Story of BigQuery (Dremel)

WebPage(URL, PageRank, Language, NumVisits, HTML)

⇒ 2 weeks back, awarded the VLDB "Test of Time" for past "10-12 years" of impact

VLDB Test of Time Award

A paper is selected from the VLDB Conference from 10 to 12 years earlier that best meets the "test of time". In picking a winner, the committee evaluates the impact of the paper. The committee especially values impact of the paper in practice, e.g., in products and services. Impact on the academic community demonstrated through significant follow-through research by the community is also valued.

To browse ealier years' VLDB Test of Time Awards, please go here.

2020

- Award Winners: Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis.
- Paper Title: Dremel: interactive analysis of web-scale datasets. Proc. VLDB Endow. 3, 1–2 (September 2010), 330–339.

⇒ Big systems perspective?

[optional: http://www.vldb.org/pvldb/vol13/p3461-melnik.pdf]

16

In this Section

(Next weeks:
Scale, Scale, Scale)

1. IO Model
2. Data layout
3. Indexing
4. Organizing Data and Indices

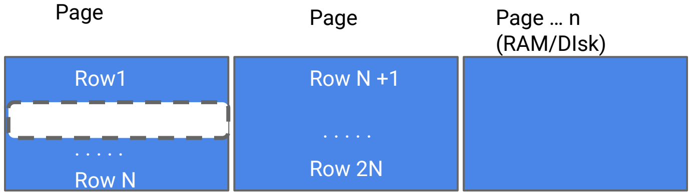
17

How to find the right data fast?

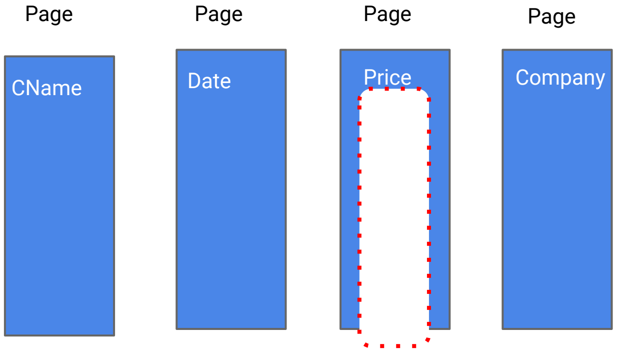
Company(<u>CName</u>, StockPrice, Date, Country)

Col3

	Company			
	CName	Date	Price	Country
Row1	AAPL	Oct1	101.23	USA
	AAPL	Oct2	102.25	USA
Row3	AAPL	Oct3	101.6	USA
	GOOG	Oct1	201.8	USA
Row5	GOOG	Oct2	201.61	USA
	GOOG	Oct3	202.13	USA
	Alibaba	Oct1	407.45	China
Row8	Alibaba	Oct2	400.23	China



Row based storage
(aka Row Store)



Column based storage
(aka Column Store)

Next: How to find AAPL Prices?

18

Why study Indexes?

1. Fundamental unit for DB performance

2. Core indexing ideas have become **stand-alone systems**
    - E.g., search in google.com
    - Data blobs in noSQL, Key-value stores
    - Embedded join processing

19

Example

Find Book in Library

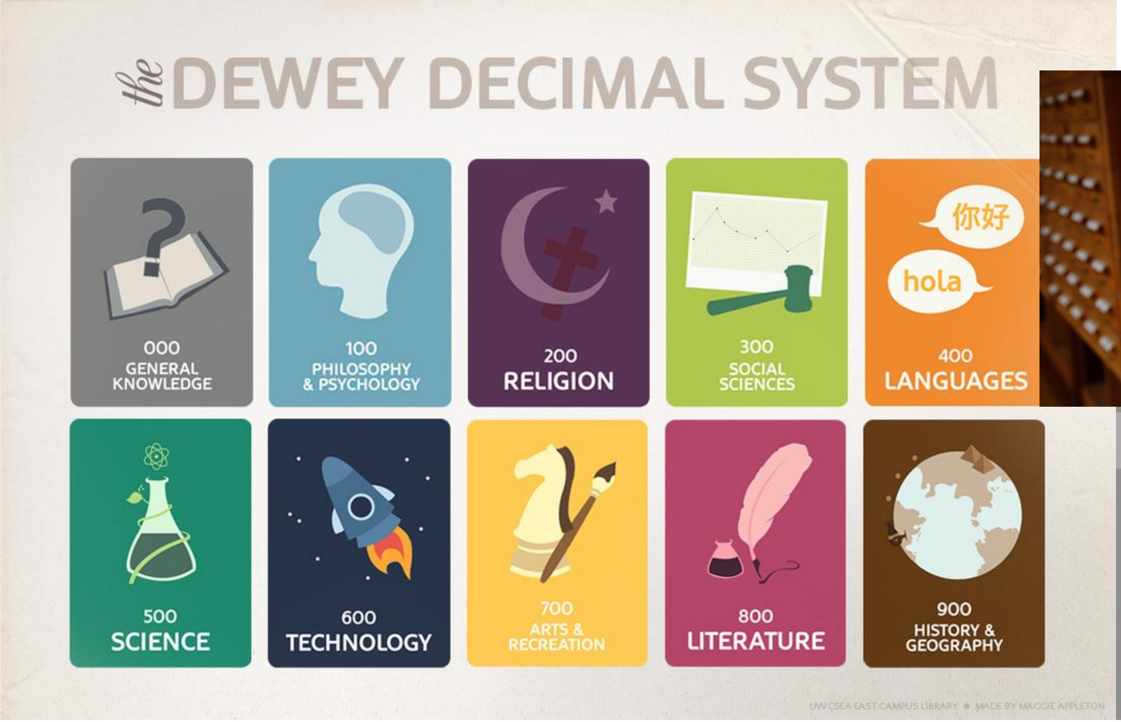
<u>Design choices?</u>

- <u>Scan</u> through each aisle
- <u>Lookup</u> pointer to book location, with librarian's organizing scheme

20

Example

Find Book in Library With Index

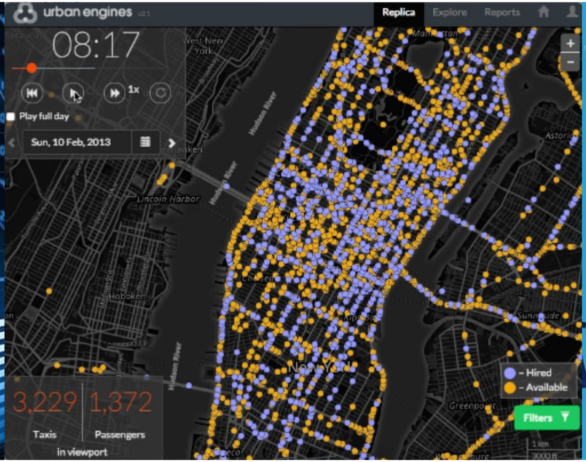


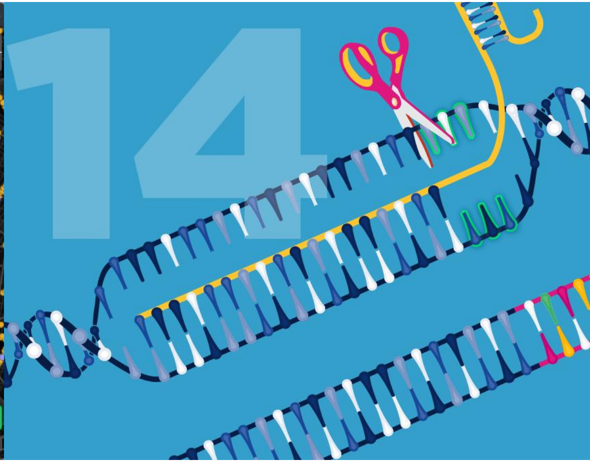
Index Cards

Algorithm for book titles

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

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

Example: Search on stocks

Company(<u>CName</u>, StockPrice, Date, Country)

Company			
CName	Date	Price	Country
AAPL	Oct1	101.23	USA
AAPL	Oct2	102.25	USA
AAPL	Oct3	101.6	USA
GOOG	Oct1	201.8	USA
GOOG	Oct2	201.61	USA
GOOG	Oct3	202.13	USA
Alibaba	Oct1	407.45	China
Alibaba	Oct2	400.23	China

```
SELECT *
FROM   Company
WHERE  CName like 'AAPL'
```

```
SELECT CName, Date
FROM   Company
WHERE  Price > 200
```

Q: On which attributes would you build indexes?

A: On as many subsets as you'd like. Look at query workloads.

Example

CName_Index

CName	Block #
AAPL	
AAPL	
AAPL	
GOOG	
GOOG	
GOOG	
Alibaba	
Alibaba	

Block #

Company			
CName	Date	Price	Country
AAPL	Oct1	101.23	USA
AAPL	Oct2	102.25	USA
AAPL	Oct3	101.6	USA
GOOG	Oct1	201.8	USA
GOOG	Oct2	201.61	USA
GOOG	Oct3	202.13	USA
Alibaba	Oct1	407.45	China
Alibaba	Oct2	400.23	China

PriceDate_Index

Date	Price	Block #
Oct1	101.23	
Oct2	102.25	
Oct3	101.6	
Oct1	201.8	
Oct2	201.61	
Oct3	202.13	
Oct1	407.45	
Oct2	400.23	

1. Index contains search key + Block #: e.g., DB block number.
   - In general, “pointer” to where the record is stored (e.g., RAM page, DB block number or even machine + DB block)
   - Index is conceptually a table. In practice, implemented very efficiently (see how soon)

2. Can have multiple indexes to support multiple search keys

Indexes (definition)

Maps search keys to sets of rows in table

- Provides efficient lookup & retrieval by search key value (much faster than scanning all rows and filtering, usually)
- Can build an index on any subset of fields in table. Advanced: build across rows, across tables
- Key operations: Lookup, Insert, Delete

An index can store

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

25

Covering Indexes

PriceDate_Index

Date	Price	Block #
Oct1	101.23	
Oct2	102.25	
Oct3	101.6	
Oct1	201.8	
Oct2	201.61	
Oct3	202.13	
Oct1	407.45	
Oct2	400.23	

An index covers *for a specific query* if the index contains all the needed attributes

I.e, *query can be answered using the index alone!*

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

Example:

```
SELECT Date, Price
FROM    Company
WHERE Price > 101
```

26

Why study Indexes?

1. Fundamental unit for DB performance

2. Core indexing ideas have become **stand-alone systems**
    - E.g., search in google.com
    - Data blobs in noSQL, Key-value stores
    - Embedded join processing

In this Section

(Next weeks:
Scale, Scale, Scale)

1. IO Model
2. Data layout
3. Indexing
4. Organizing Data and Indices

28

Big Scale

Roadmap

Hashing

Sorting

Counting

Hashing-Sorting-Counting solves "all" known data scale problems :=)

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

THE BIG IDEA

Note

Works for Relational, noSQL

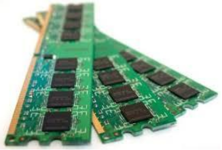
(e.g. mySQL, postgres, BigQuery, BigTable, MapReduce, Spark)

29

Big Scale

Roadmap

Primary data structures/algorithms

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

Hashing

Recall: Hashing

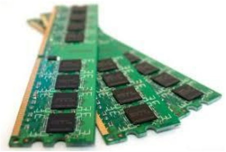
“AAPL” → **Hash Function** → 0x5e7A

“GOOG” → **Hash Function** → 0x6b7F

<Oct1, 201.73, “GOOG”> → **Hash Function** → 0x7c6D

<32.31, -122.23, “9:01 pm> (lat, lng, time)

0x5e7A → **Location Func** → Location in RAM (e.g., % NumPage)



0x5e7A → **Location Func** → Location in <Machine, Disk, RAM>

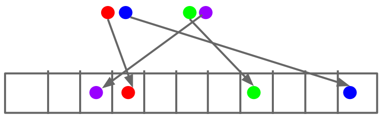
- Magic of hashing:
  - A hash function h_B maps into [0,B-1], nearly uniformly
  - Also called sharding function

- A hash collision is when x != y but $h_B(x) = h_B(y)$
  - Note however that it will never occur that x = y but $h_B(x) \mathrel{!=} h_B(y)$

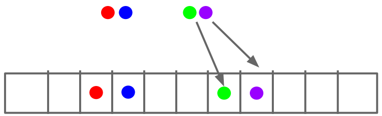
Hashing ideas for scale

- Idea: Multiple hash functions (uncorrelated to spread data)
  - $h_i(x), h_{i+1}(x), h_{i+2}(x), h_{i+3}(x), \ldots$

- Idea: Locality sensitive hash functions (for high dimensional data)
  - Special class of hash functions to keep spread ‘local’



Regular hash functions
(spread all over)

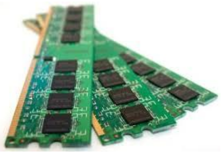


Locality Sensitive Hash (LSH) functions
(spread in closer buckets, with high probability)

Big Scaling (with Indexes)

Roadmap

Primary data structures/algorithms



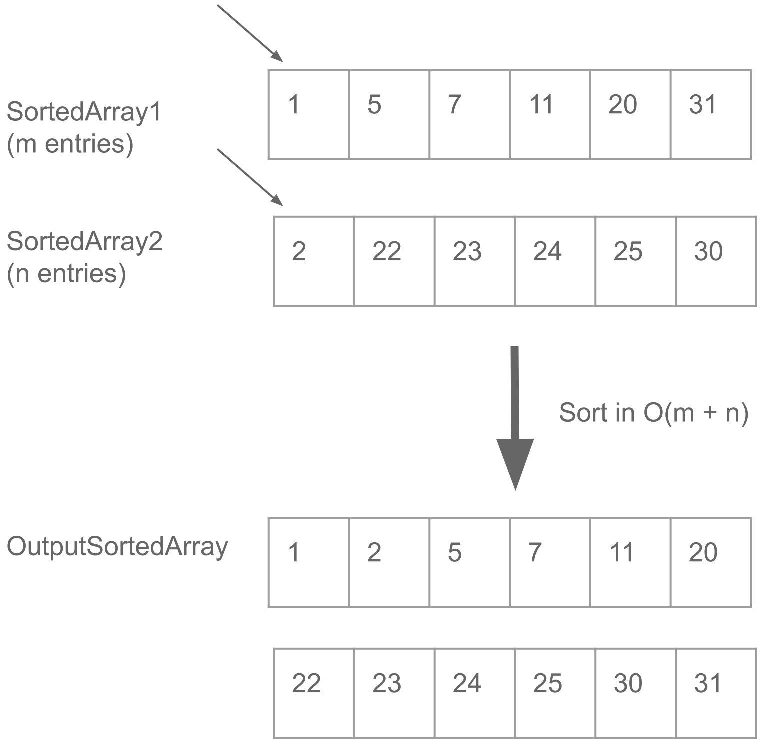
Hashing	Sorting	Counting
HashTables $(hash_i(key)$ --> value)	BucketSort, QuickSort MergeSort2Lists, MergeSort	HashTable + Counter $(hash_i(key)$ --> <count>)
	MergeSortedFiles, MergeSort	

Sorting 1:
External Merge Algorithm

35

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:

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

Then:

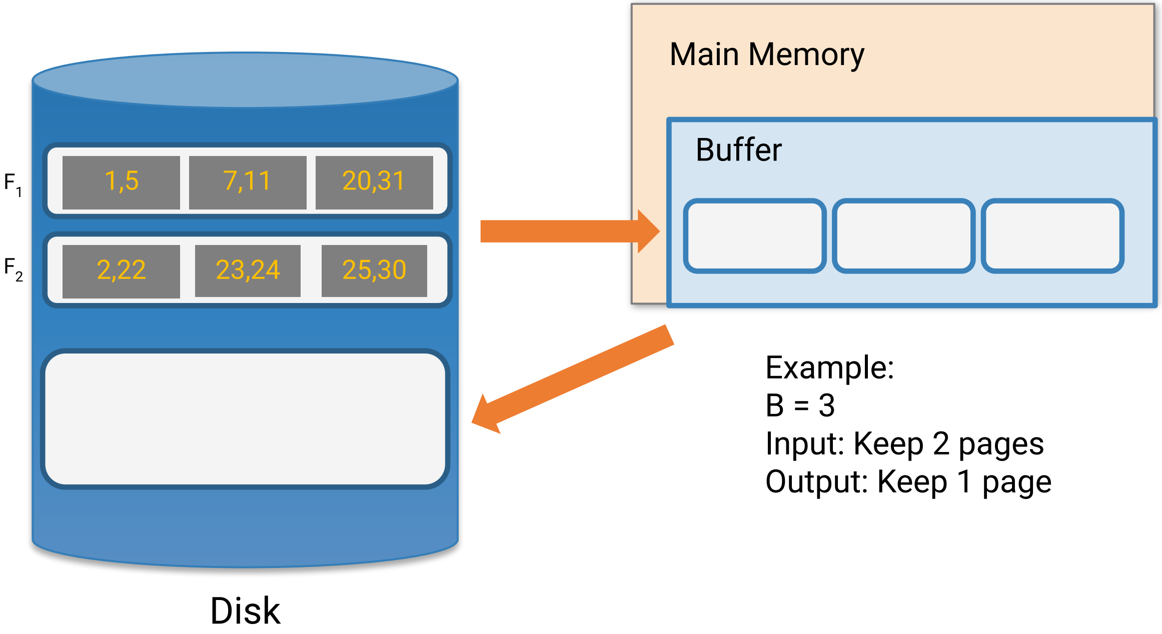
$$Min(A_1, B_1) \le A_i$$
$$Min(A_1, B_1) \le B_j$$

for i=1….N and j=1….M

External Merge Algorithm

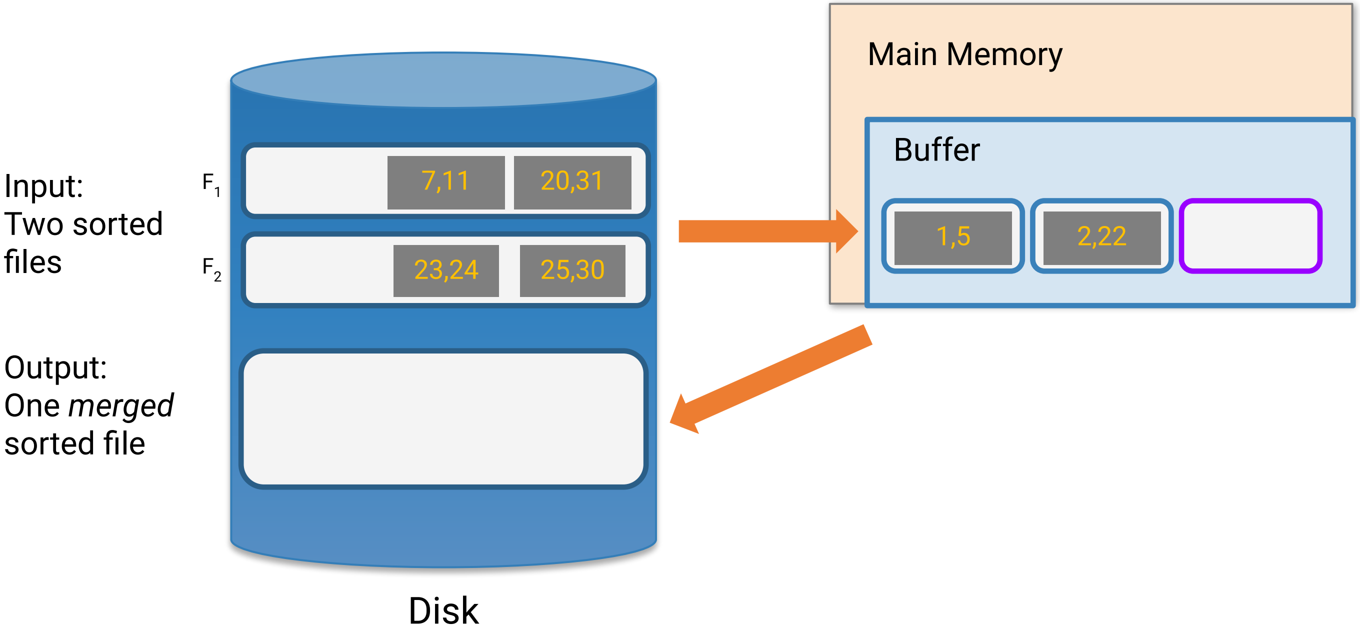
Input:
Two sorted files

Output:
One *merged* sorted file



Example:
B = 3
Input: Keep 2 pages
Output: Keep 1 page

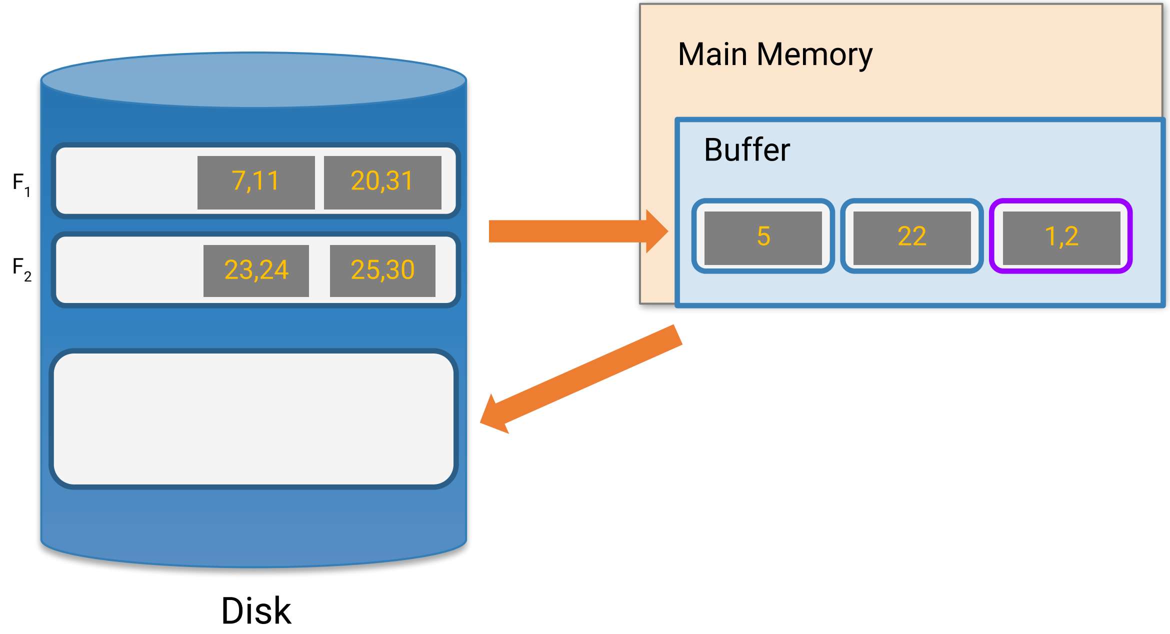
External Merge Algorithm



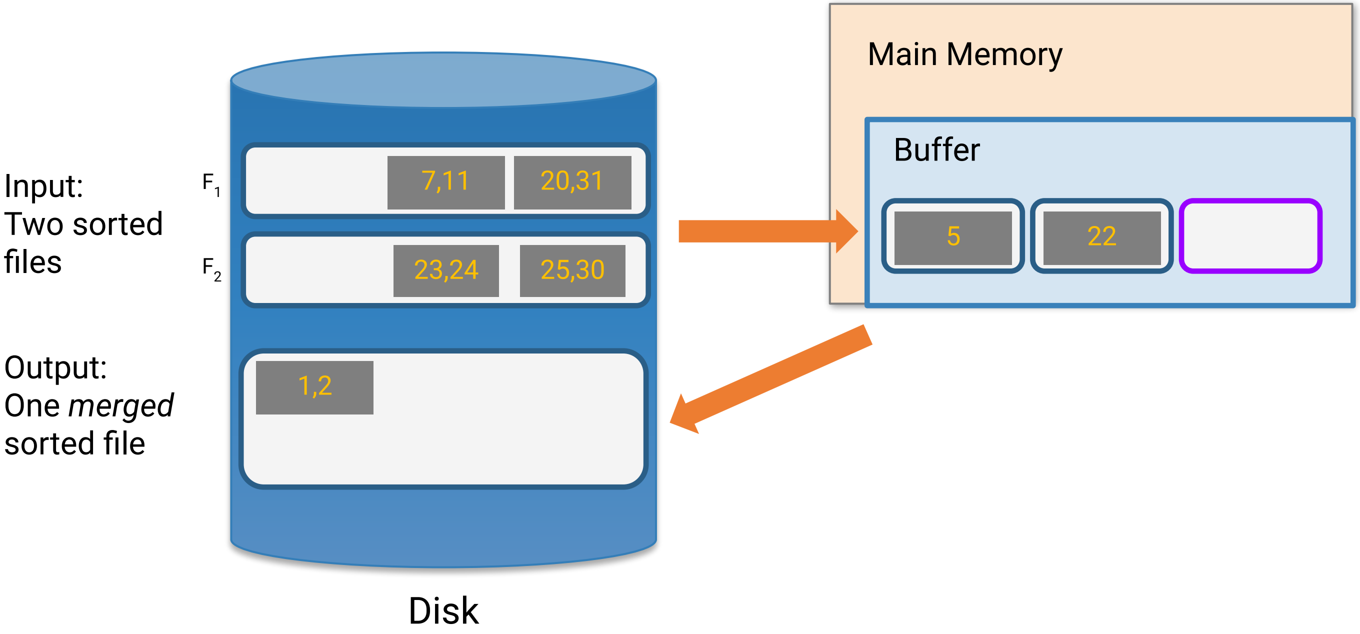
External Merge Algorithm

Input:
Two sorted
files

Output:
One *merged*
sorted file



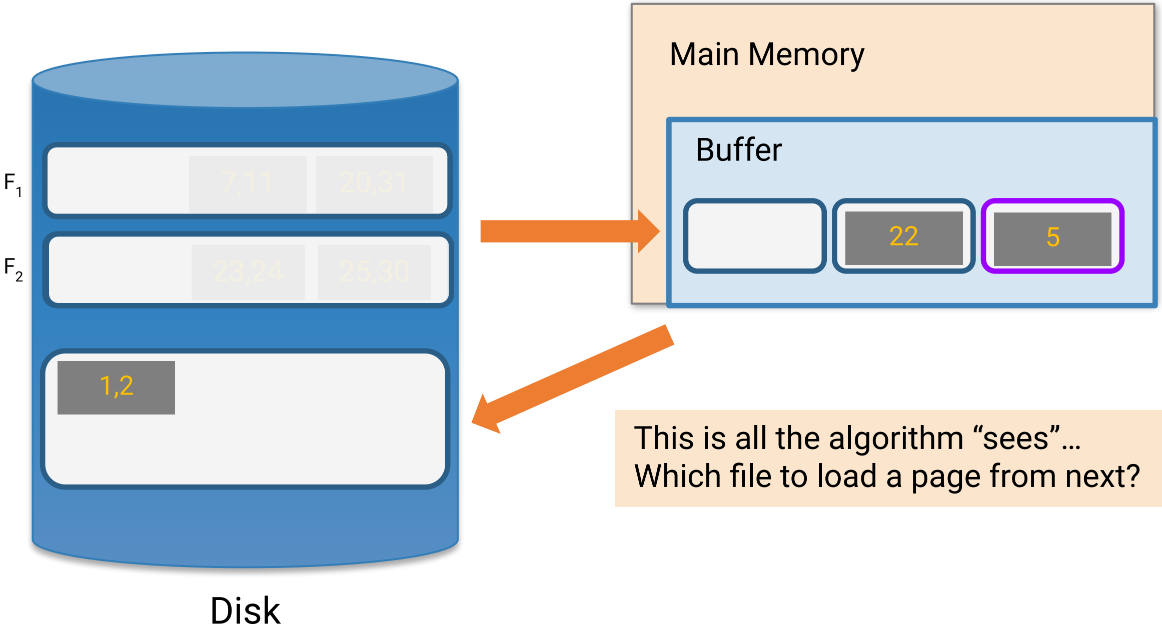
External Merge Algorithm



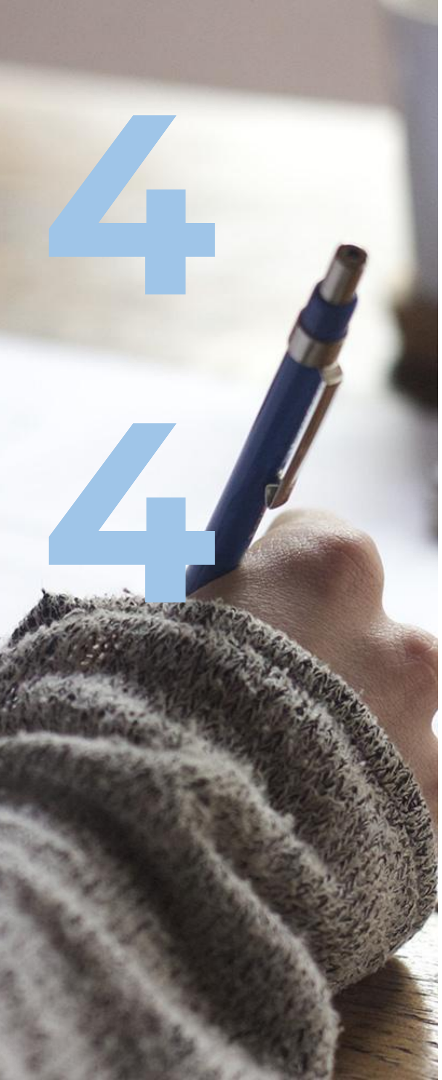
External Merge Algorithm

Input:
Two sorted files

Output:
One *merged* sorted file



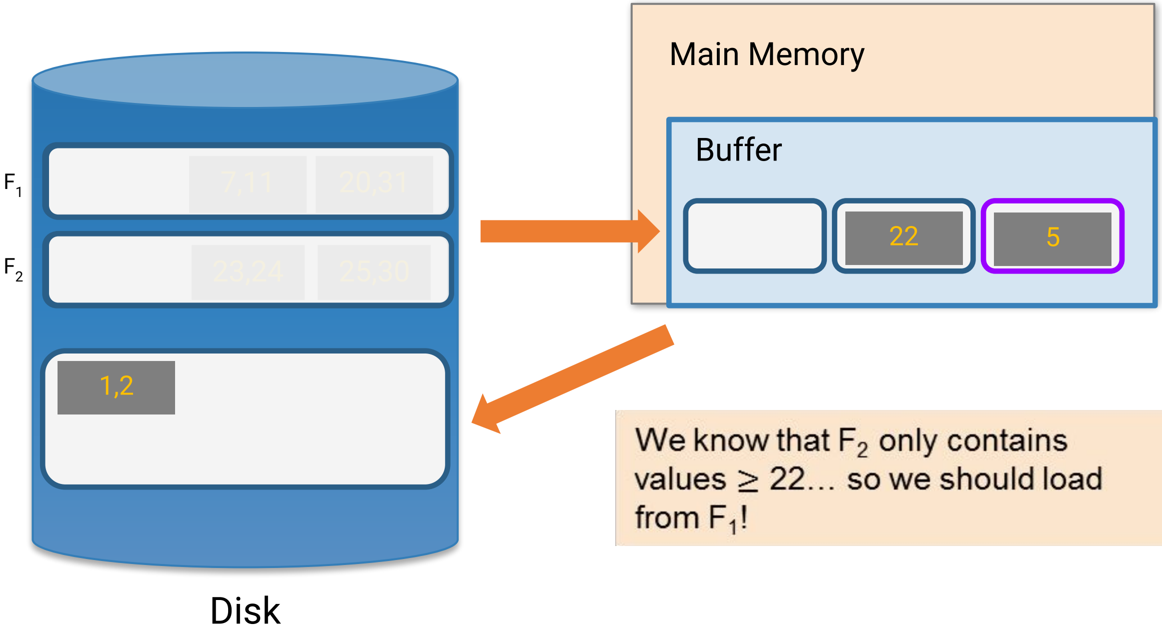
This is all the algorithm “sees”…
Which file to load a page from next?



External Merge Algorithm

Input:
Two sorted files

Output:
One *merged* sorted file

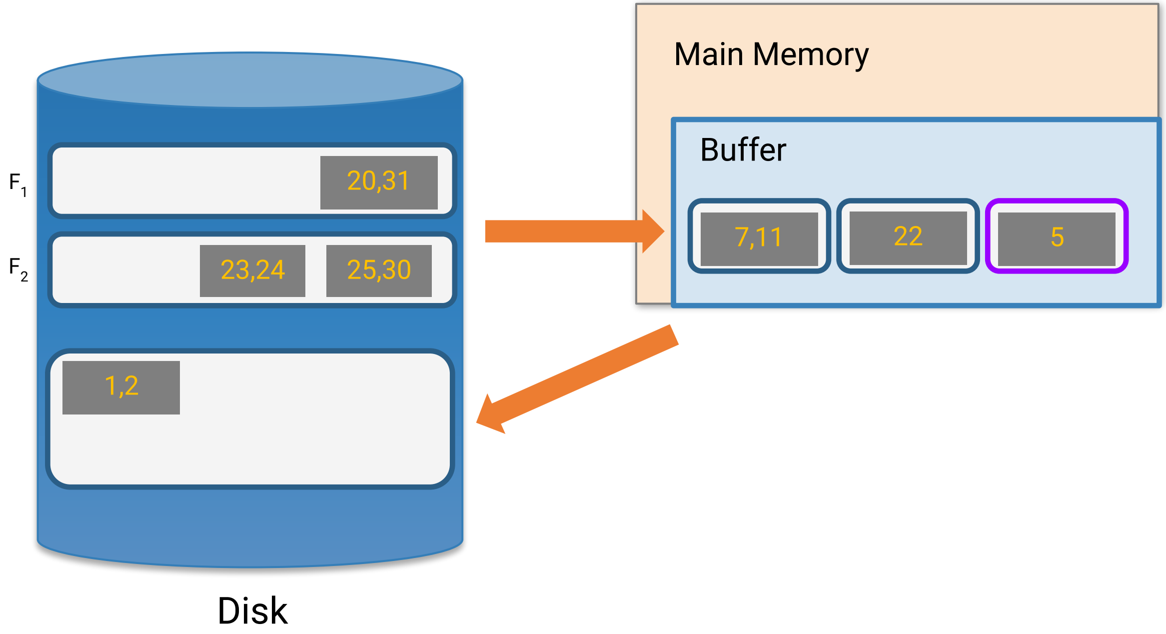


We know that F_2 only contains values $\geq$ 22… so we should load from F_1!

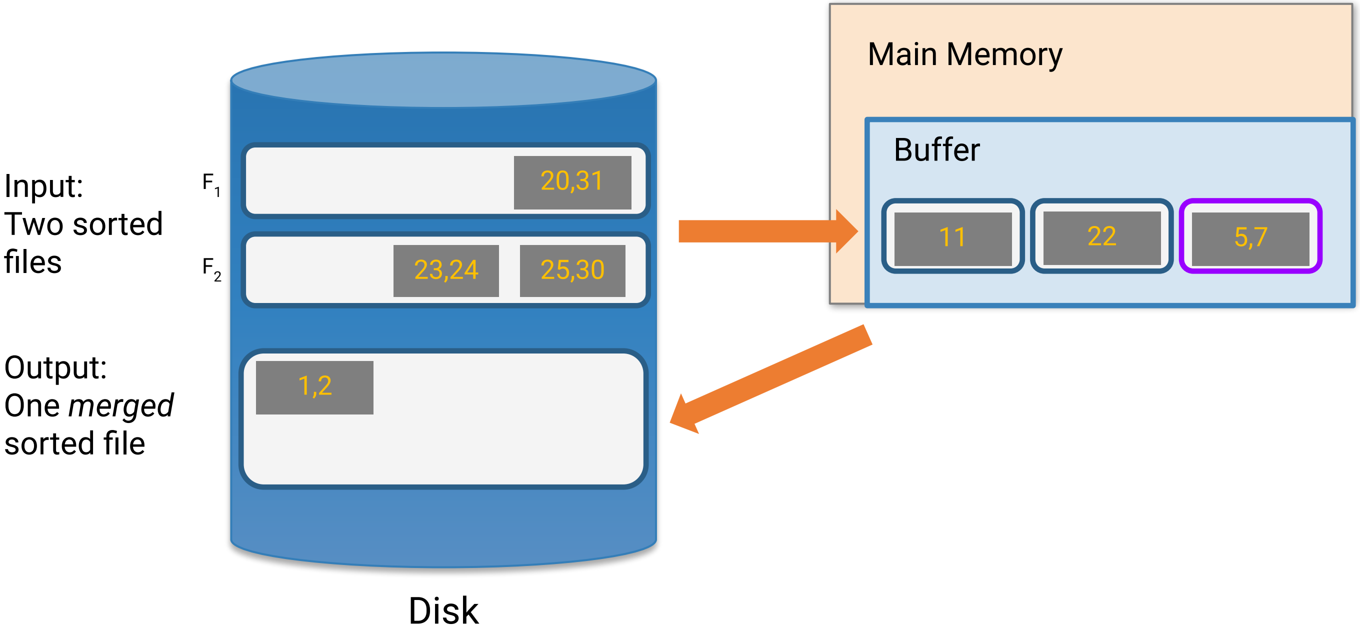
External Merge Algorithm

Input:
Two sorted files

Output:
One *merged* sorted file

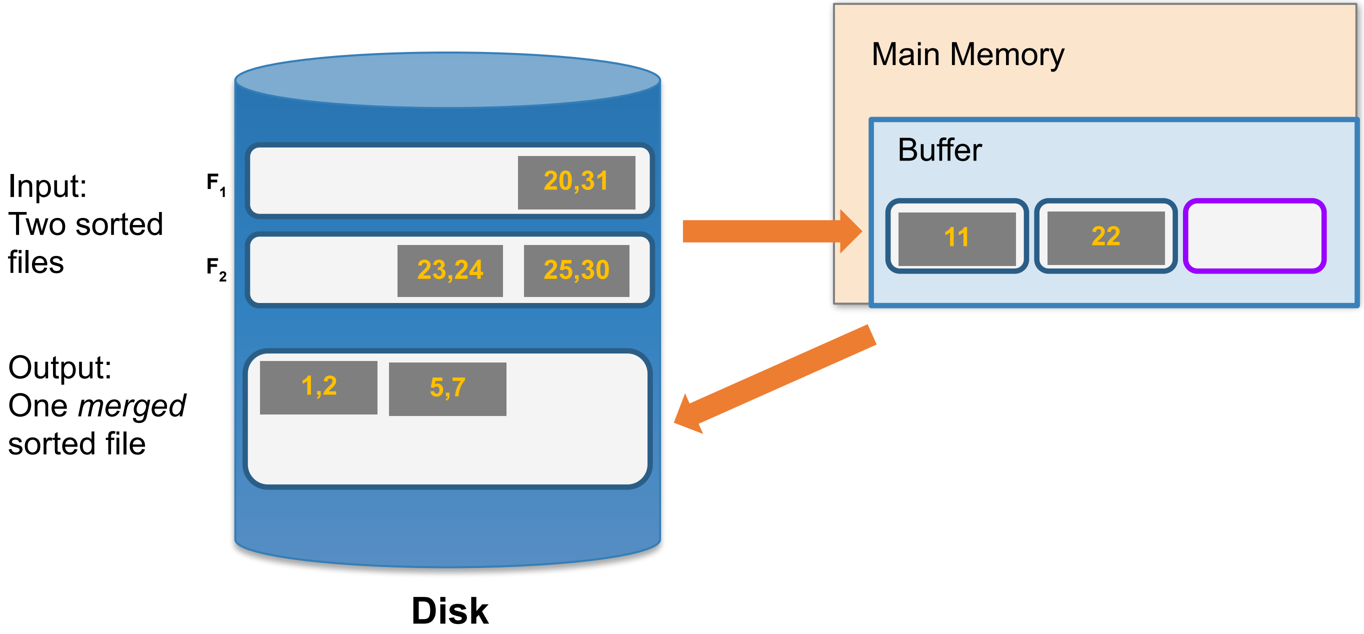


External Merge Algorithm

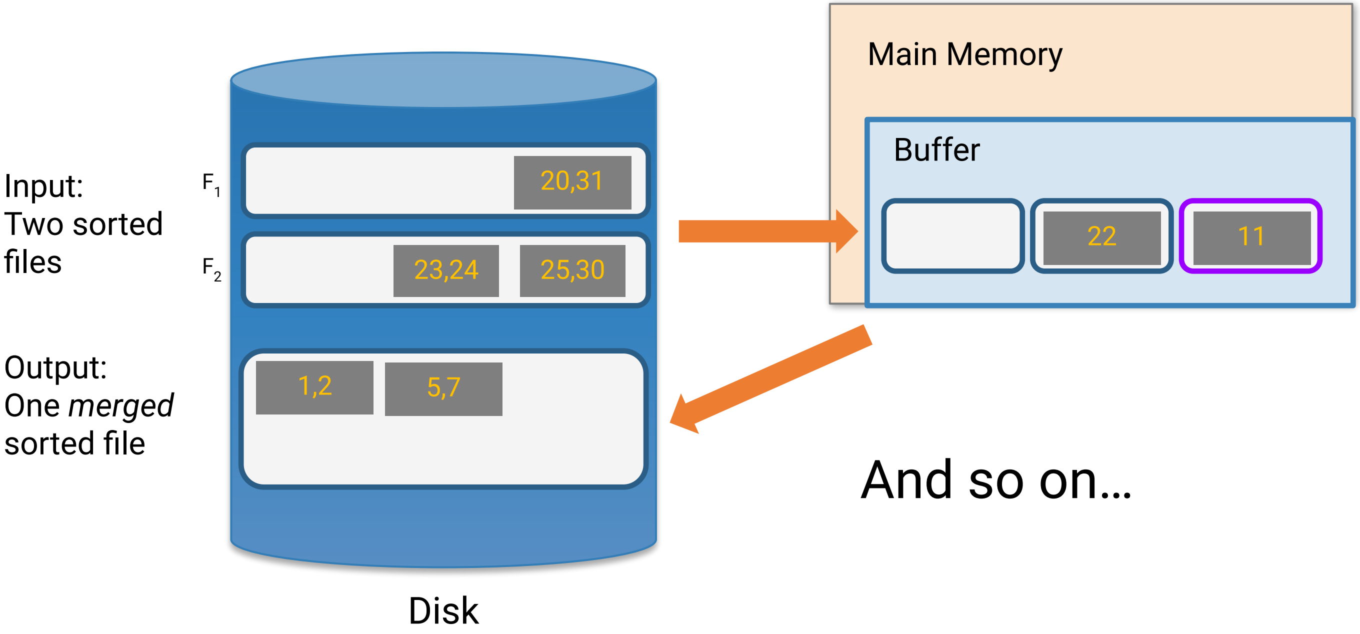


47

External Merge Algorithm



External Merge Algorithm



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

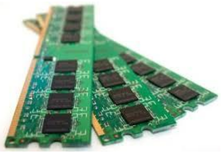
51

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

?????

In this Section

(Next weeks:
Scale, Scale, Scale)

1. IO Model
2. Data layout
   - row vs column storage
3. Indexing
4. Organizing Data and Indices
   - Hashing, Sorting, Counting