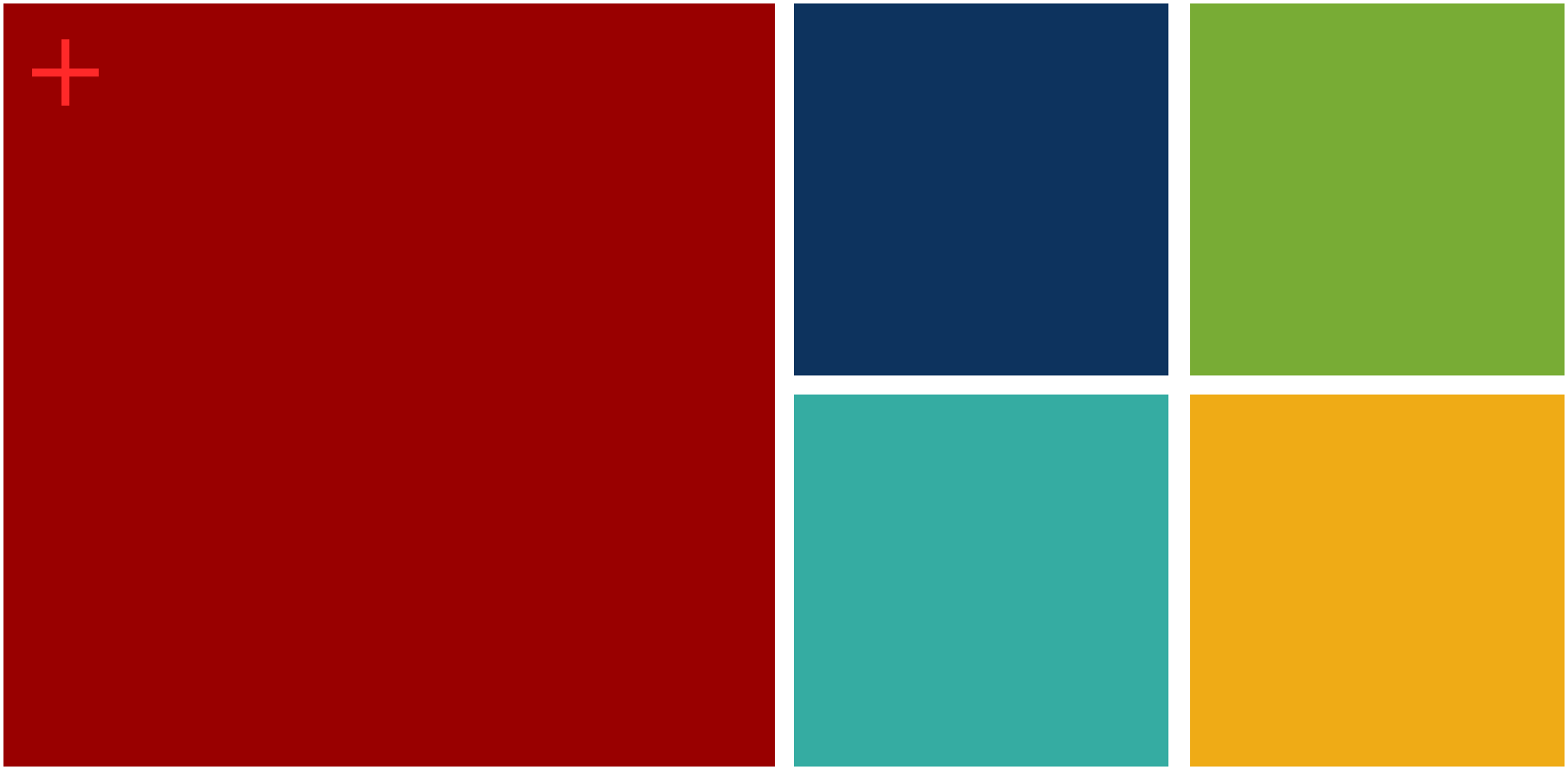


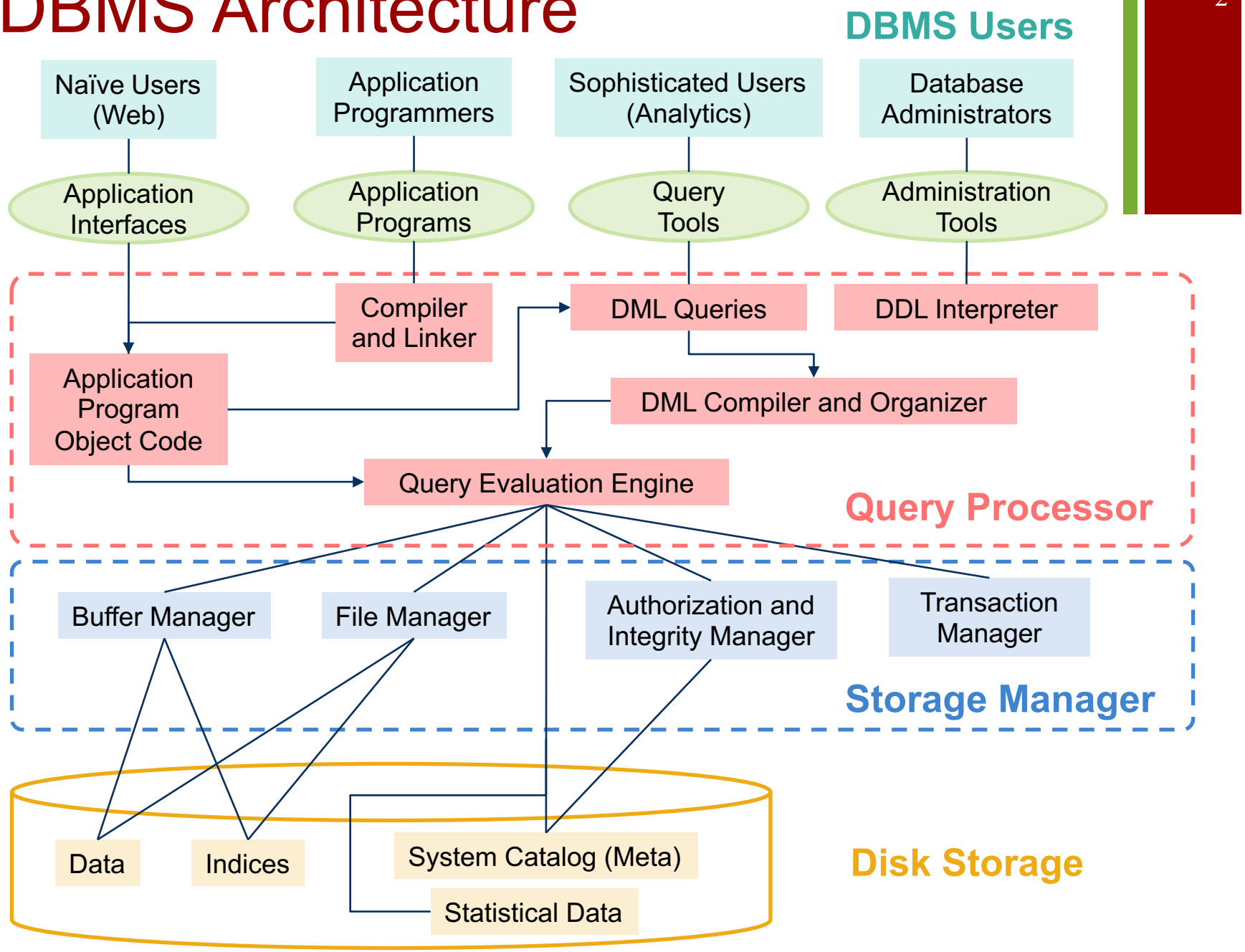
DSAA 5012 Advanced Database Management for Data Science

Storage and File Structure

Fall 2026

The Hong Kong University of Science and Technology (Guangzhou)

+ DBMS Architecture



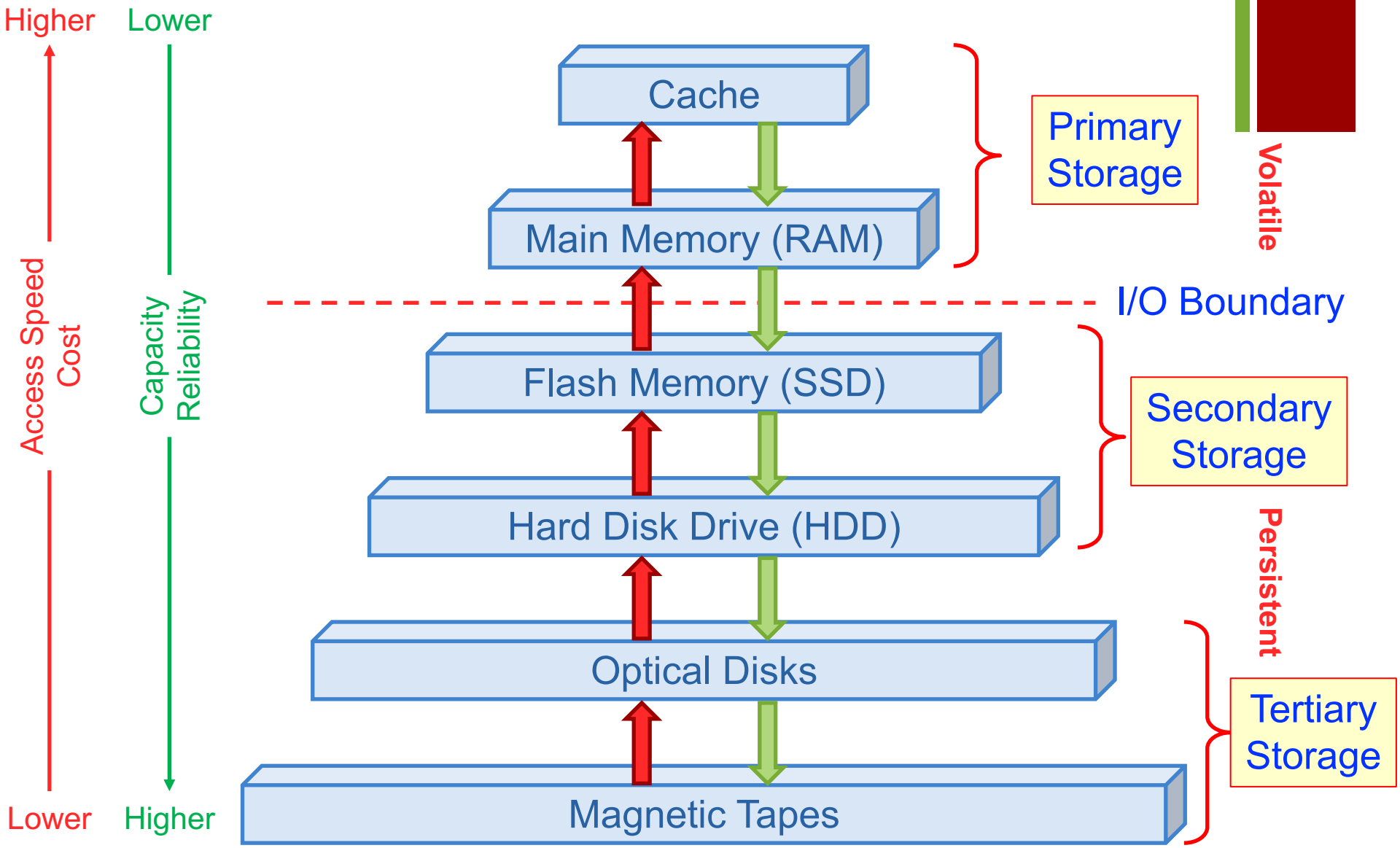
+ Outline

1. **Physical Storage Media Overview**
   - HDD | I/O Cost | RAID levels
2. **Database Buffer Manager**
3. **Record Organization**
   - Fixed-length records | Free-lists | Slotted Page
4. **File Organizations**
   - Heap File | Sequential File | Hash File

Advanced

5. **Column-Oriented Storage Formats**
   - Storage Model | Encoding | Nested Data | Parquet | ORC
6. **SIMD-Optimized Advanced Compression**
   - BtrBlocks | FastLanes | BitWeaving

+ Storage Device Hierarchy



+ Access Time

- Latency Numbers Every Programmer Should Know

1 ns	L1 Cache	1 second
4 ns	L2 Cache	4 seconds
100 ns	DRAM	100 seconds
16,000 ns	SSD	4.4 hours
2,000,000 ns	HDD	3.3 weeks
50,000,000 ns	Network Storage	1.5 years
1,0000,000,000 ns	Tape Archives	31.7 years

+ Magnetic Tape

- Still in use???
  - Cheap per GB: 2 cent
  - Better durability: 50 years or longer
  - Suit for backup of PB historical data
    - Telecom
  - Energy Efficient
- Fast sequential read/write
  - LTO 8: 12TB 360MB/s
  - LTO 9: 25TB 708MB/s
  - LTO 10: 48TB 1100MB/s





IBM TS4500 Tape Library

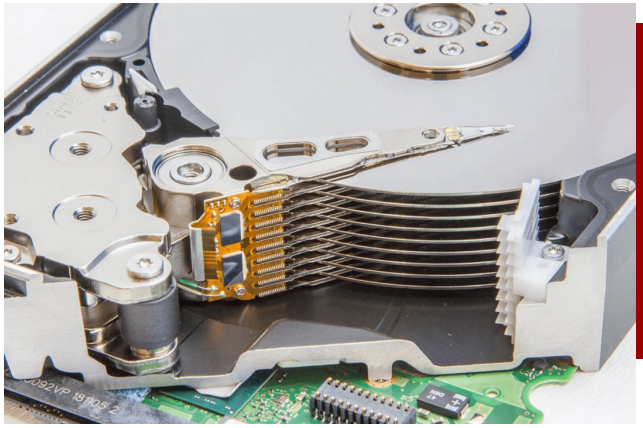
Next-generation cloud storage library.

- Maximum capacity with LTO-8 / TS1160: 278 PB / 351 PB
- Drive Type(s): LTO and/or TS1100
- Maximum number of drives: 128



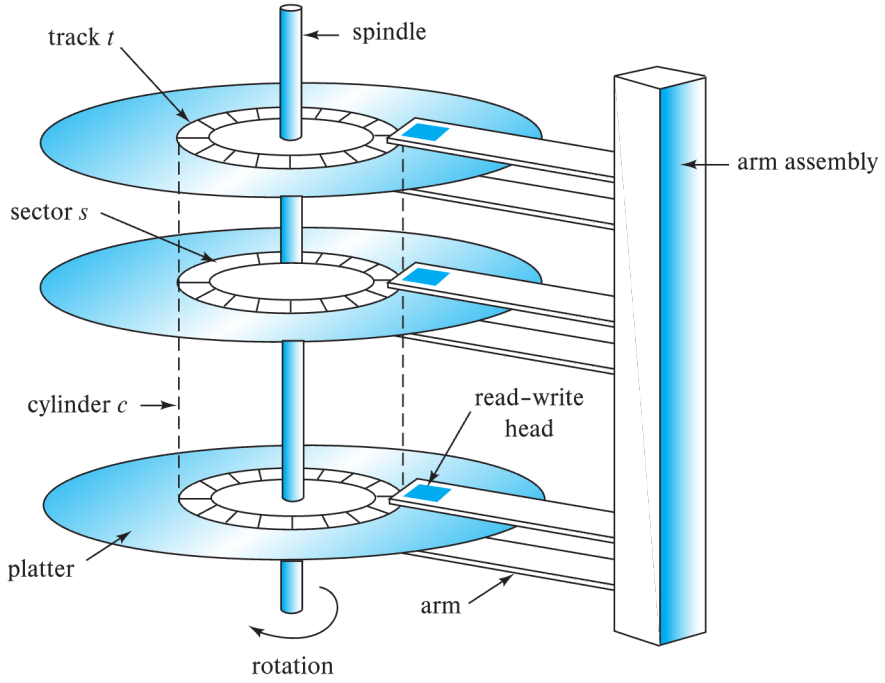


+ Hard Disk Drive (HDD)



- The platters spin
- The arm assembly moves in or out to position a head on a desired track
- Tracks under all the heads make a (imaginary) cylinder
- Only one head reads/writes at any one time since it is hard for all heads to line up on a track exactly
- The page/block size is often the unit of retrieval and is a multiple of the sector size (which is fixed)

The most common secondary storage device



A sector is the smallest unit that can be physically read/written

Disk READ/WRITE operations are much slower than in-memory operations

+ Disk Page Access

- Time to access (read/write) a page on an HDD consists of
  1. **Seek Time**
     - Time to move the arms to position the disk head on a track. Seek time varies from about 4 to 15 msec.
  2. **Rotational Delay (Latency)**
     - Time to wait for the page (sector) to rotate under the head
     - Rotational delay varies from 2 to 7 msec.
  3. **Transfer Time**
     - Time to move data to/from the disk surface.
     - The transfer rate is about 1 msec. per 4KB page.

Seek time and rotational delay dominate

Sequential I/O is much faster than random I/O

The key to better I/O performance: reduce seek time/rotational delay!

+ RAID

- **RAID: Redundant Arrays of Independent Disks**
  - Disk organization techniques that manage a large numbers of disks, providing a view of a single disk of
    - **High capacity** and **High speed** by using multiple disks in parallel
    - **High reliability** by storing data redundantly, so that data can be recovered even if a disk fails
- The chance that some disk out of a set of *N* disks will fail is much higher than the chance that a specific single disk will fail
  - A system with 100 disks, each with MTTF (mean time to failure) of 100,000 hours (approx. 11 years), will have a system MTTF of 1000 hours (approx. 41 days)
  - Techniques for using redundancy to avoid data loss are critical with large numbers of disks

Improvement in Reliability via Redundancy

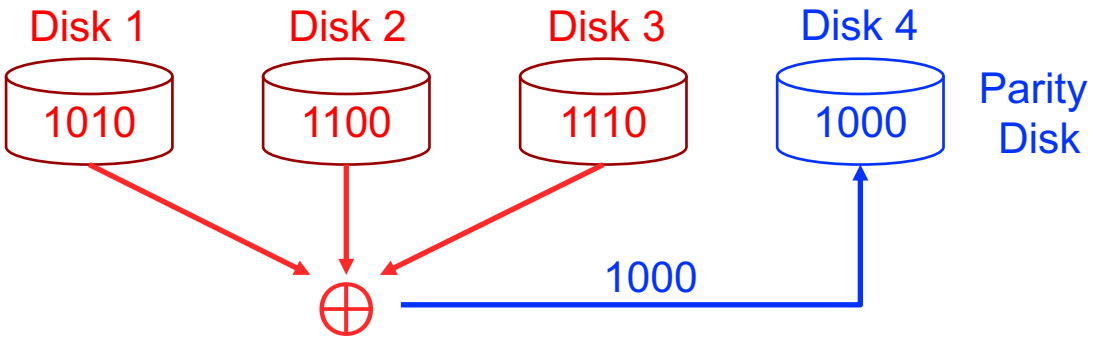
- **Redundancy** – Store extra information that can be used to rebuild information lost in a disk failure
  - Mirroring (or Shadowing)
  - Duplicate every disk
    - Logical disk consists of two physical disks
  - Every write is carried out on both disks
    - Reads can take place from either disk
  - If one disk in a pair fails, data still available in the other
  - Data loss would occur only if a disk fails, and its mirror disk also fails before the system is repaired
    - Probability of combined event is very small
    - Except for dependent failure modes such as fire or building collapse or electrical power surges
- Mean time to data loss depends on mean time to failure, and mean time to repair

Improvement in Performance via Parallelism

- Two main goals of parallelism in a disk system:
  1. Load balance multiple small accesses to increase throughput
  2. Parallelize large accesses to reduce response time
- Improve transfer rate by striping data across multiple disks
- **Bit-level striping** – split the bits of each byte across multiple disks
  - In an array of eight disks, write bit i of each byte to disk i
  - Each access can read data at eight times the rate of a single disk
  - But seek/access time worse than for a single disk
    - Bit level striping is not used much any more
- **Block-level striping** – with n disks, block i of a file goes to disk $(i \bmod n) + 1$
  - Requests for different blocks can run in parallel if the blocks reside on different disks
  - A request for a long sequence of blocks can utilize all disks in parallel

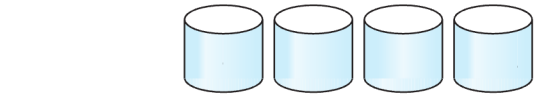
+ RAID Levels

- **Parity Blocks**: Parity block *j* stores XOR of bits from block *j* of each disk
  - When writing data to a block *j*, parity block *j* must also be computed and written to disk
    - Can be done by using old parity block, old value of current block and new value of current block (2 block reads + 2 block writes)
    - Or by recomputing the parity value using the new values of blocks corresponding to the parity block
      - More efficient for writing large amounts of data sequentially
  - To recover data for a block, compute XOR of bits from all other blocks in the set including the parity block

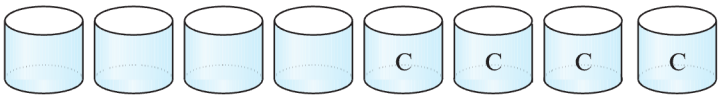


+ RAID Levels

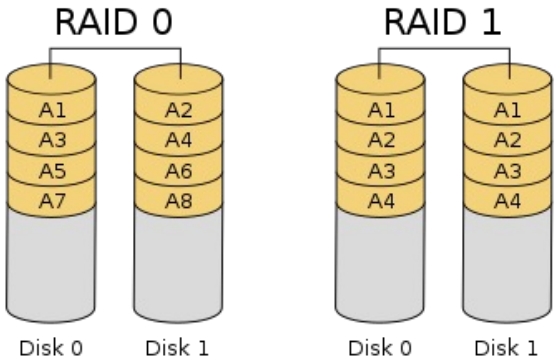
- Schemes to provide redundancy at lower cost by using disk striping combined with parity bits
  - Different RAID organizations, or RAID levels, have differing cost, performance and reliability characteristics
- **RAID Level 0**: Block striping, Non-redundant
  - Used in high-performance applications where data loss is not critical
- **RAID Level 1**: Mirrored disks with block striping
  - Offers best write performance
  - Popular for applications such as storing log files in a database system



(a) RAID 0: nonredundant striping

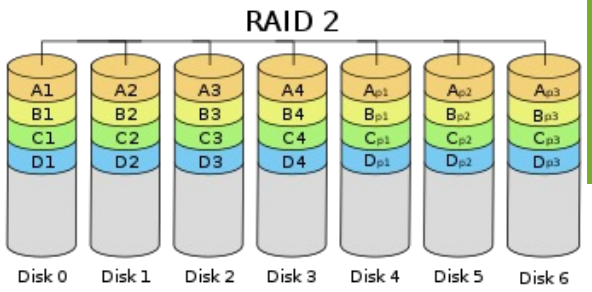


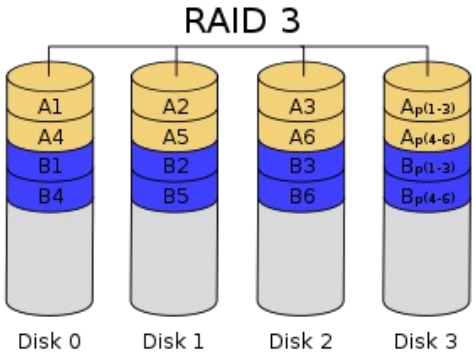
(b) RAID 1: mirrored disks

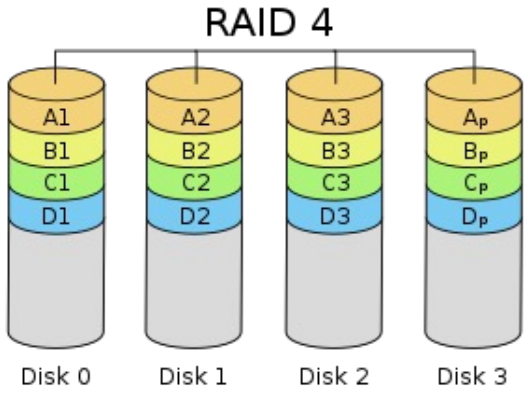


RAID Levels (Outdated Ones)

- **RAID Level 2**
  - Bit-level striping
  - Hamming code for error correction
- **RAID Level 3**
  - Byte-level striping
  - One parity disk
- **RAID Level 4**
  - Block-level striping
  - One parity disk





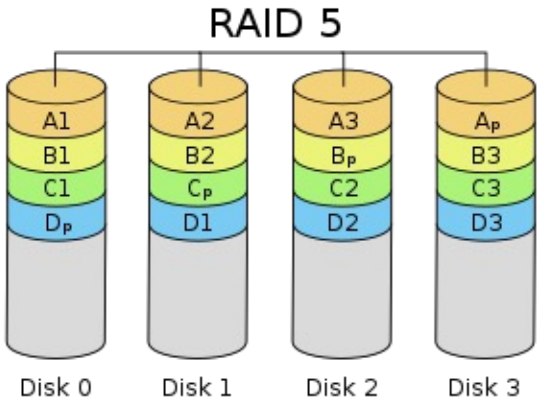


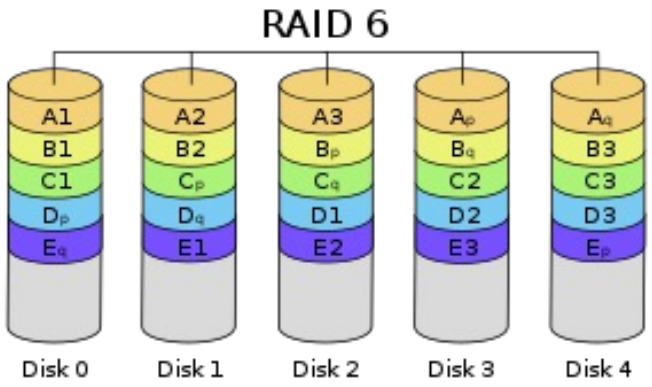
Parity disk gets much higher write load than other disks and becomes a bottleneck

+ RAID Levels

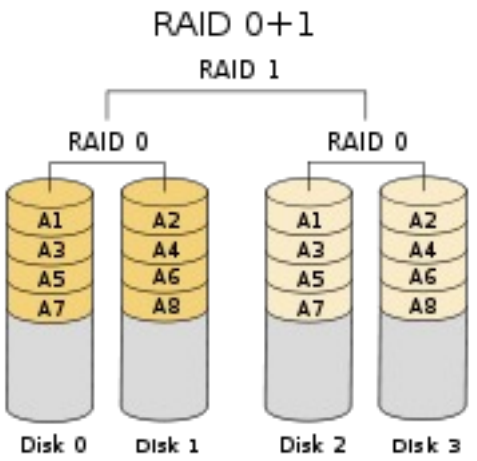
- **RAID Level 5:** Block-Interleaved Distributed Parity;
  - Partitions data and parity among all *N* + 1 disks, rather than storing data in *N* disks and parity in 1 disk
  - Example: with 4 disks, parity block for n^{th} set of blocks is stored on disk (*n mod* 4) + 1, with the data blocks stored on the other 3 disks.
  - Block writes occur in parallel if the blocks and their parity blocks are on different disks

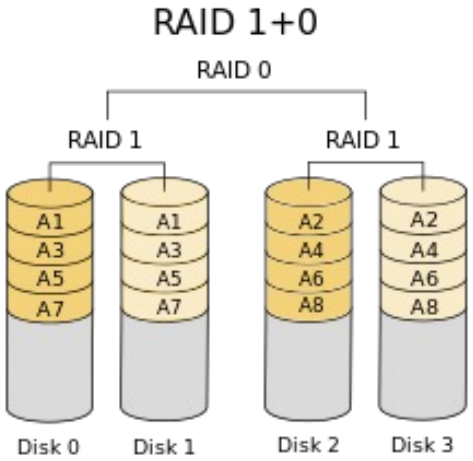
- **RAID Level 6**: P+Q Redundancy scheme
  - Similar to Level 5, but stores two error correction blocks (P, Q) instead of single parity block to guard against multiple disk failures.
  - Better reliability than Level 5 at a higher cost
    - Becoming more important as storage sizes increase

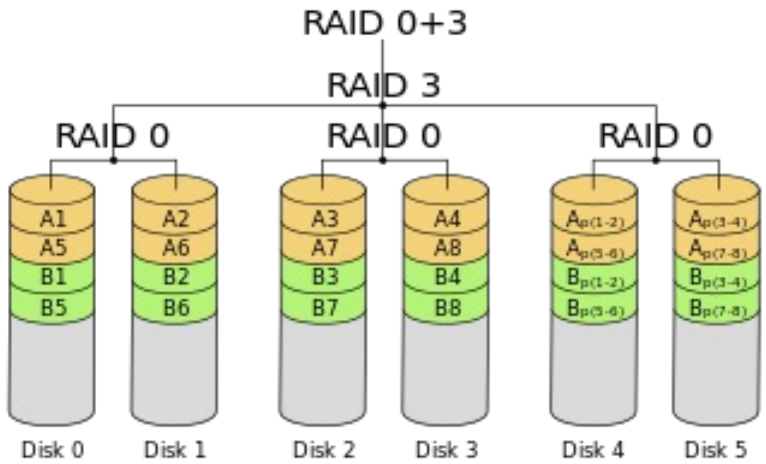


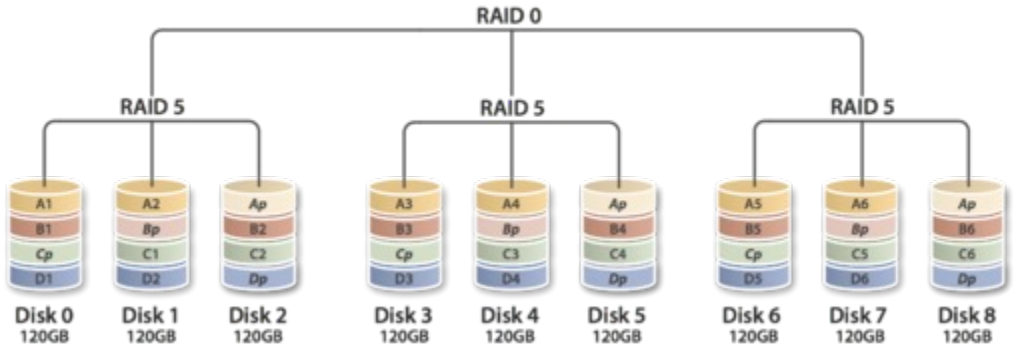


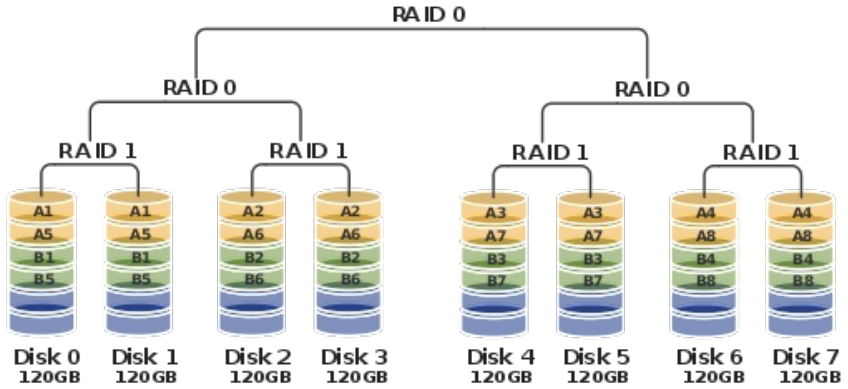
+ RAID Nested Levels

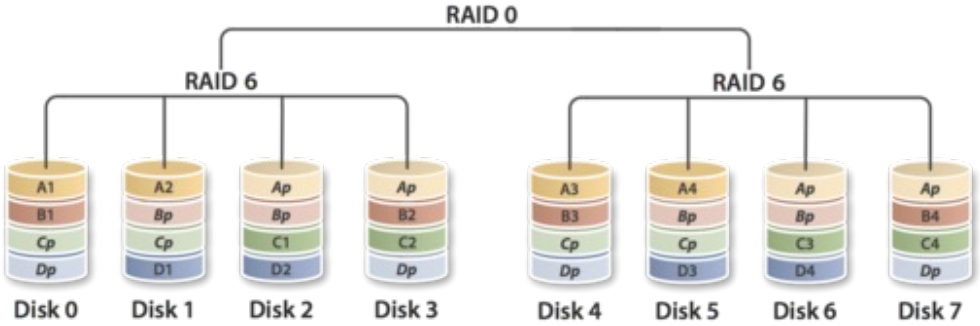












+ Choice of RAID Levels

- Factors in choosing RAID level
  - Monetary cost
  - Performance: Number of I/O operations per second, and bandwidth during normal operation
  - Performance during failure
  - Performance during rebuild of failed disk
    - Including time taken to rebuild failed disk
- Choices
  - RAID 0 is used only when data safety is not important
    - When data can be recovered quickly from other sources
  - Level 1 provides much better write performance than level 5
    - Level 5 requires at least 2 block reads and 2 block writes to write a single block, whereas Level 1 only requires 2 block writes
  - Level 1 had higher storage cost than level 5
  - Level 5 is preferred for applications where writes are sequential and large (many blocks), and need large amounts of data storage
  - RAID 1 is preferred for applications with many random/small updates
  - Level 6 gives better data protection than RAID 5 since it can tolerate two disk (or disk block) failures
    - Increasing in importance since latent block failures on one disk, coupled with a failure of another disk can result in data loss with RAID 1 and RAID 5

Outline

1. **Physical Storage Media Overview**
   - HDD | I/O Cost | RAID levels
2. **Database Buffer Manager**
3. **Record Organization**
   - Fixed-length records | Free-lists | Slotted Page
4. **File Organizations**
   - Heap File | Sequential File | Hash File
5. **Column-Oriented Storage Formats**
   - Storage Model | Encoding | Nested Data | Parquet | ORC
6. **SIMD-Optimized Advanced Compression**
   - BtrBlocks | FastLanes | BitWeaving

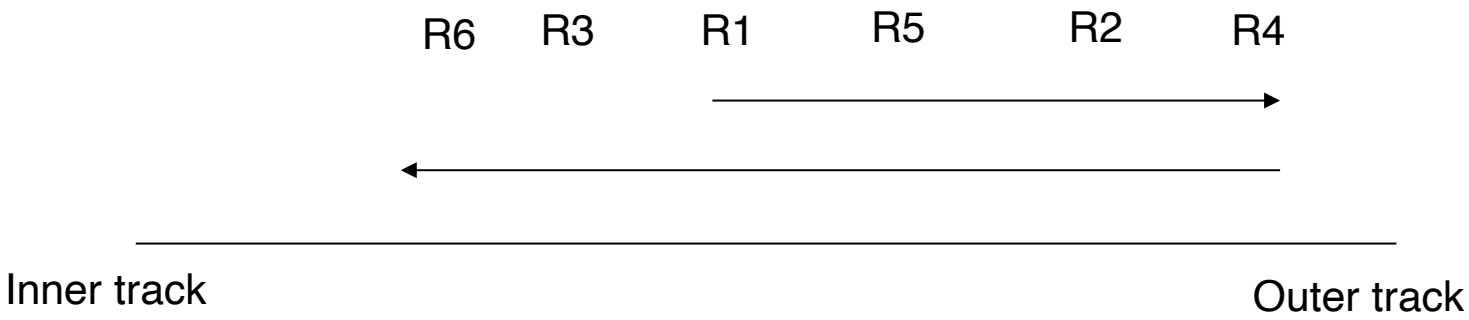
Storage Access

- To reduce seek time/rotational delay, database systems try to minimize page transfers (read/write) between disk and memory
- The number of disk accesses can be reduced by keeping as many pages as possible in main memory
- **Buffer:** The portion of main memory available to store copies of disk pages
  - Main memory space for storing disk pages is limited!
- **Buffer Manager:** The subsystem responsible for managing the memory buffer space
  - The goal of the buffer manager is to minimize disk accesses

Data must be in RAM for an application to operate on it!

Optimization of Disk-Block Access

- **Buffering:** In-memory buffer to cache disk blocks
- **Read-ahead:** Read extra blocks from a track in anticipation that they will be requested soon
- **Disk-arm-scheduling:** Algorithms re-order block requests so that disk arm movement is minimized
  - **Elevator algorithm**



Why Buffer Management?

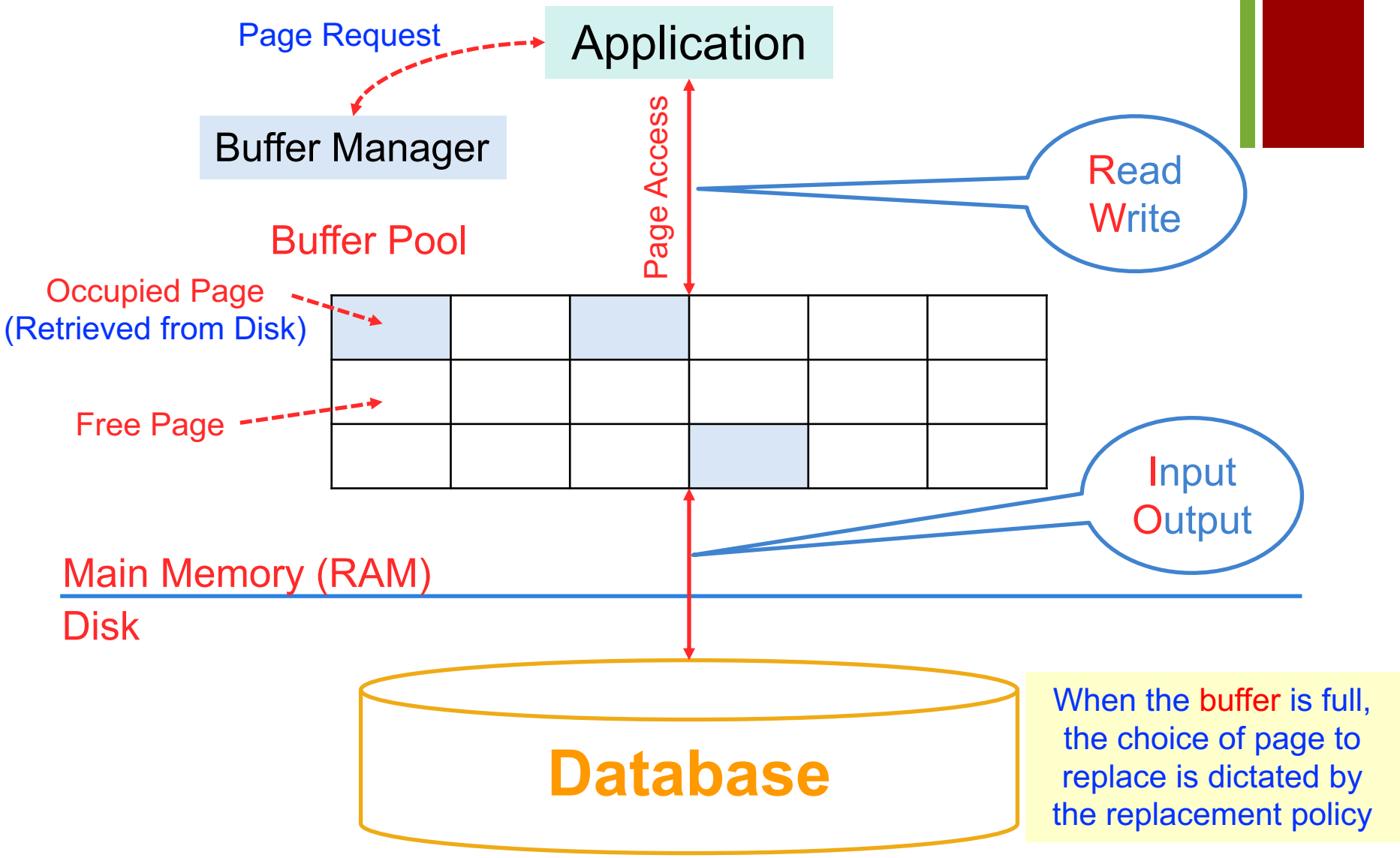
- Ordinary applications rely on OS for IO
  - Memory mapping **mmap**
    - Store the contents of a file into the address space of a program
    - OS is responsible for moving file pages in and out of memory, so the DBMS doesn't need to worry about it
    - Or does it???
  - **Problems**
    1. Transaction Safety: OS can flush dirty pages at any time
    2. I/O Stalls: DBMS doesn't know which pages are in memory
       - The OS will stall a thread on page fault
    3. Performance Issues
  - **Solutions**
    1. **madvise**: Tell the OS how you expect to read certain pages
    2. **mlock**: Tell the OS that memory ranges cannot be paged out
    3. **msync**: Tell the OS to flush memory ranges out to disk

Why Buffer Management?

- DBMS (almost) always wants to control things itself and can do a better job than the OS
  1. Flushing dirty pages to disk in the correct order
  2. Specialized prefetching
  3. Buffer replacement policy
  4. Thread/process scheduling

OS is not our friend!

+ Buffer Management



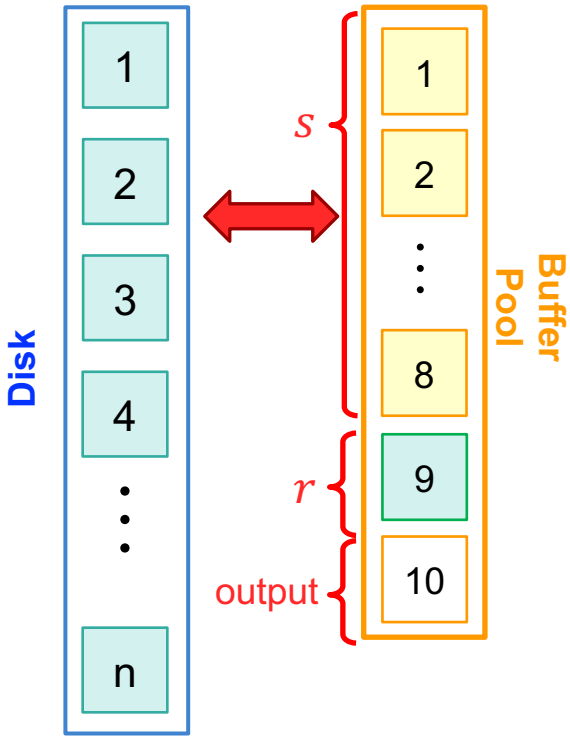
When the buffer is full, the choice of page to replace is dictated by the replacement policy

+ Buffer Replacement Policies

- Most operating systems replace (evict) the page *Least Recently Used* (LRU strategy).
- LRU can be a bad strategy for certain access patterns involving repeated scans of data
  - Example: Compute the join of two relations r and s using a nested loop
  - Best in this case: Replace the page most recently used (MRU strategy)
- A DBMS usually has its own buffer manager that uses statistical information regarding the probability that a request will reference a particular relation or page

Suppose memory can hold 10 pages and relation s needs 8 pages.

For each page in r
  For each page in s
    relation r ⋈ relation s



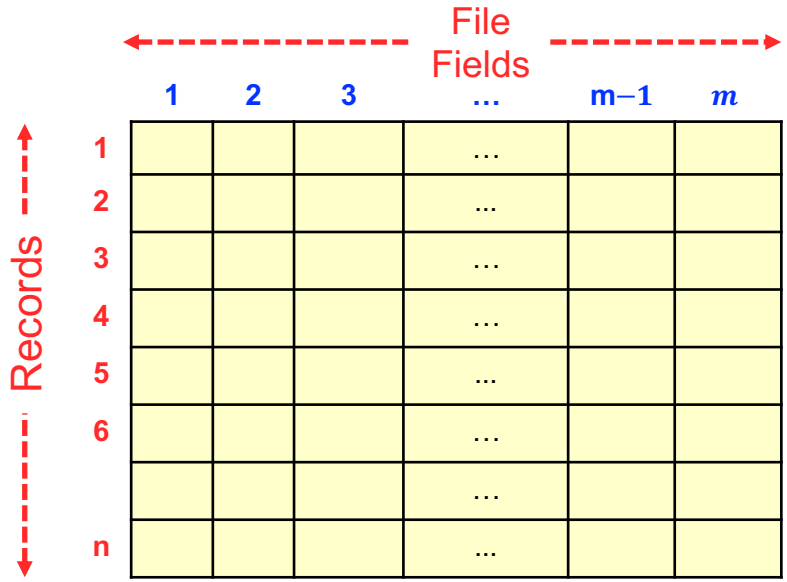
+ Outline

1. **Physical Storage Media Overview**
   - HDD | I/O Cost | RAID levels
2. **Database Buffer Manager**
3. **Record Organization**
   - Fixed-length records | Free-lists | Slotted Page
4. **File Organizations**
   - Heap File | Sequential File | Hash File
5. **Column-Oriented Storage Formats**
   - Storage Model | Encoding | Nested Data | Parquet | ORC
6. **SIMD-Optimized Advanced Compression**
   - BtrBlocks | FastLanes | BitWeaving

+ Record Organization

- Record organization is the organization of data items, which usually represent attributes, into physically stored records

- A database is stored as
  - A collection of *files* where
  - Each file is a sequence of *records* and
  - Each record is a sequence of *fields* that occupy several bytes



- Most common organization
  - Assume the record size is fixed
    - Each record occupies a fixed number of bytes
  - Each file has records of one type only
  - Different files are used for different relations

  - This organization is the easiest to implement

Fixed-Length Records: Relative Location

- In each page, store record i starting from byte $n \times (i - 1)$, where n is the size of each record in bytes
- Record access is simple, but records may cross (span) pages
  - Normally file systems do not allow records to cross page boundaries (unspanned)
  - Consequently, there may be some unused space at the end of a page.

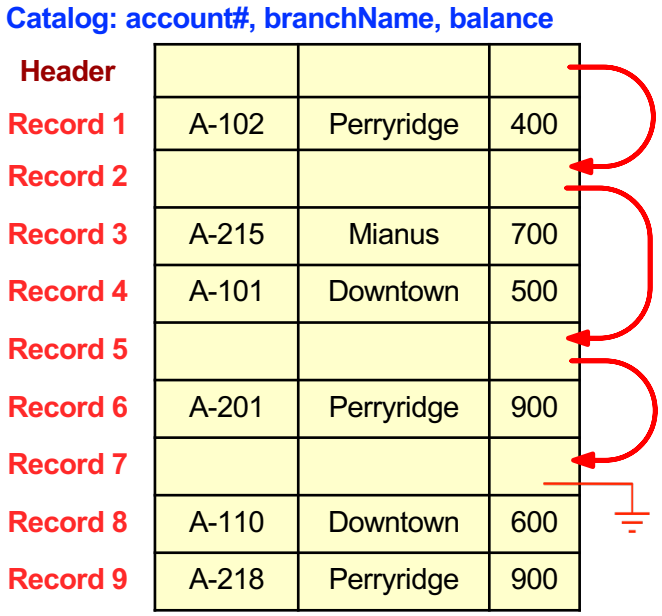
Catalog: account#, branchName, balance

Record 1	A-102	Perryridge	400
Record 2	A-305	Round Hill	350
Record 3	A-215	Mianus	700
Record 4	A-101	Downtown	500
Record 5	A-222	Redwood	700
Record 6	A-201	Perryridge	900
Record 7	A-217	Brighton	750
Record 8	A-110	Downtown	600
Record 9	A-218	Perryridge	900

- To reclaim space when record i is deleted, shift records up
- However, moving records inside a page is not good when records are pointed to by:
  1. Other records (e.g., foreign keys)
  2. Index Entries

+ Fixed-Length Records: Free Lists

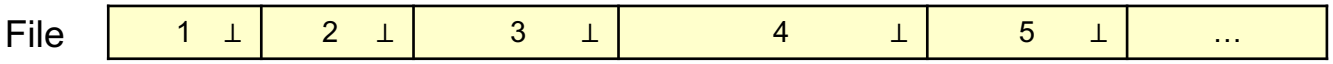
- Do not move records in a page; instead, store the address of the first deleted record in the file header (the first record)
- Use the first deleted record to store the address of the second deleted record and so on
- Can think of these stored addresses as pointers since they "point" to the location of a record



- This is a more space-efficient representation since we can reuse space for *normal* attributes of free records to store the pointers.
  - No pointers are stored in records being used

Variable-Length Records: Byte-String Representation

- Variable-length records arise in database systems due to
  1. Storage of multiple types of records in a file
  2. Records that allow variable lengths for one or more fields
     - Varchar data type
  3. Records that allow repeating fields (not allowed in relational DBMSs)



- Simple (but bad) solution: **byte-string representation**
  - Store each record one after the other as a string of bytes
  - Attach a special *end-of-record* (⊥) symbol to the end of each record
  - Problems with
    - Record deletion (results in fragmentation of free space)
    - File growth (may require movement of records if ordered)

Variable-Length Records: Embedded Identification

- Precede fields with metadata
  - Attribute name
- Similar to approaches used for semi-structured data
  - XML, JSON
- Requires extra storage space, but efficient if record size is variable or data is missing (e.g., not applicable)
- Similar problems as byte-string representation

Record 1	account#	A-102	branchName	Perryridge	balance	400
Record 2	account#	A-305	branchName	Round Hill	balance	350
Record 3	account#	A-215	branchName	Mianus	balance	700
Record 4	account#	A-101	branchName	Downtown	balance	500
Record 5	account#	A-222	branchName	Redwood	balance	700
Record 6	account#	A-201	branchName	Perryridge	balance	900

+ Variable-Length Records: Reserved Space

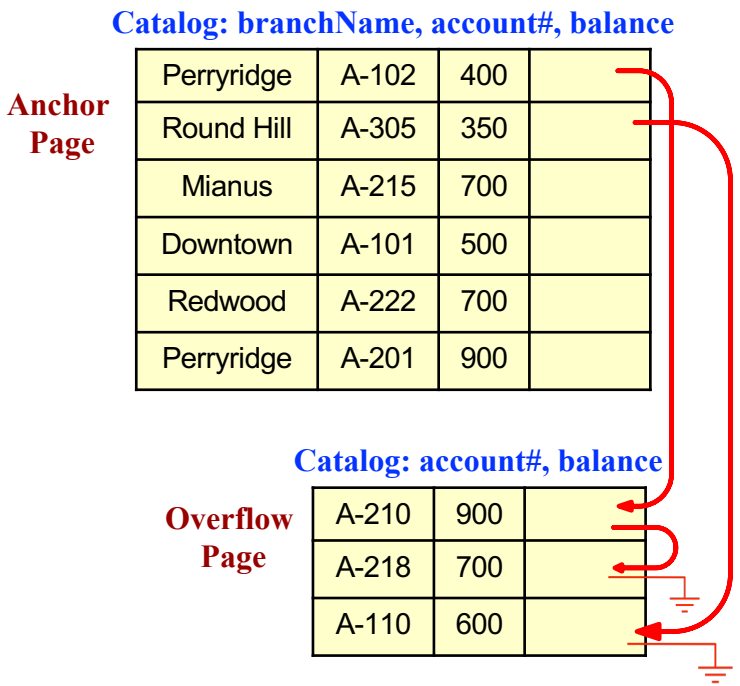
- Use fixed-length records of a known maximum length
- Unused space in shorter records is filled with a *null* or *end-of-record* symbol
- Can result in much empty, unused space in a file if record lengths vary widely

Catalog: branchName, account#, balance, account#, balance, …

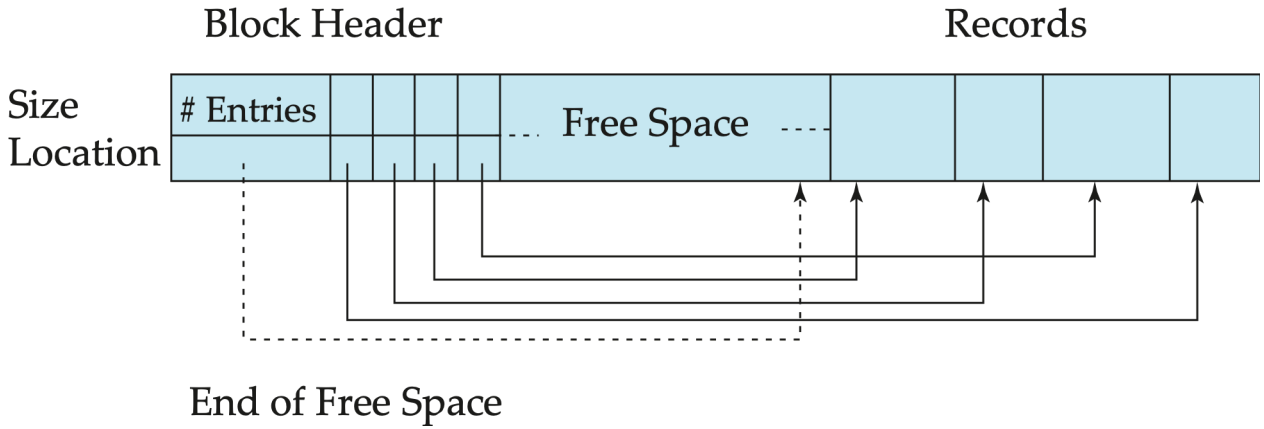
Record 1	Perryridge	A-102	400	A210	900	A218	700
Record 2	Round Hill	A-305	350	⊥	⊥	⊥	⊥
Record 3	Mianus	A-215	700	⊥	⊥	⊥	⊥
Record 4	Downtown	A-101	500	A110	600	⊥	⊥
Record 5	Redwood	A-222	700	⊥	⊥	⊥	⊥
Record 6	Perryridge	A-201	900	⊥	⊥	⊥	⊥

Variable-Length Records: Pointer Method

- Useful for certain types of records with repeating attributes
- Requires two kinds of pages in a file:
  1. **Anchor page**
     - Contains the first records of a chain
  2. **Overflow page**
     - Contains records other than those that are the first records of a chain
- Pointers
  - Link related records together



+ Variable-Length Records: Slotted-Page Structure



- The page header contains:
  - The number of record entries
  - The end location of the free space in the page
  - The location and size of each record
- Records can be moved within a page to keep them contiguous with no empty space between them
  - The page header must be updated when a record is moved.
- References to records do not point directly to the records instead, they point to the entry for the record in the page header.

+ Outline

1. Physical Storage Media Overview
   - HDD | I/O Cost | RAID levels
2. Database Buffer Manager
3. Record Organization
   - Fixed-length records | Free-lists | Slotted Page
4. **File Organizations**
   - Heap File | Sequential File | Hash File
5. **Column-Oriented Storage Formats**
   - Storage Model | Encoding | Nested Data | Parquet | ORC
6. **SIMD-Optimized Advanced Compression**
   - BtrBlocks | FastLanes | BitWeaving

File Storage

- The DBMS stores a database as one or more files on disk typically in a proprietary format
  - The OS doesn't know anything about the contents of these files
  - Early systems in the 1980s used custom file systems on raw block storage
    - Some "enterprise" DBMSs still support this
    - Most newer DBMSs do not do this
- The **Storage Manager** is responsible for maintaining a database's files
  - Some do their own scheduling for reads and writes to improve spatial and temporal locality of pages
- It organizes the files as a collection of pages
  - Tracks data read/written to pages
  - Tracks the available space

Database Pages

- There are three different notions of "pages" in a DBMS:
  - Hardware Page (usually 4KB)
  - OS Page (usually 4KB, x64 2MB/1GB)
  - Database Page (512B-32KB, 64MB for Big Data Systems)
- A hardware page is the largest block of data that the storage device can guarantee failsafe writes
  - Larger DB Page for sequential read
  - But suffers from write

4KB







8KB





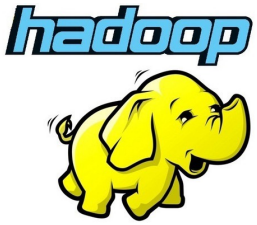




16KB



64MB



File Organization

- **Search Key**: One or more attributes by which records are retrieved

  - The records in a physical file are usually organized to facilitate efficient retrieval on a search key

  - If the search key is a:
    - Primary or candidate key ⟹ At most one record is retrieved
    - Non-key ⟹ Multiple records can be retrieved

  - Most common file organizations are
    1. Heap
    2. Sequential
    3. Hash (random)

File Blocking Factor

The blocking factor of a file r, bf_r , is the number of records that fit in a page and is equal to (for unspanned records)

[# bytes per page / # bytes per record]

Consequently, the number of pages needed to store a file is equal to

[# records / bf_r]

+ Exercise 1

bf = [# bytes per page / # bytes per record]
#pages = [# records / bf_r]

- A Student file has 20,000 records of fixed-length. Assume the page size is 512 bytes and each record has the following fields:
  - name: 30 bytes, studentId: 8 bytes, address: 40 bytes, phone: 8 bytes,
  - birthdate: 8 bytes, gender: 1 byte, majorDeptCode: 4 bytes, minorDeptCode: 4 bytes,
  - classCode: 4 bytes, degreeProgram: 3 bytes.
  - An additional byte is used as a deletion marker.

a) What is the record size in bytes?

Record Size: 30 + 8 + 40 + 8 + 8 + 1 + 4 + 4 + 4 + 3 + 1 = 111 bytes

b) What is the blocking factor $bf_{Student}$?

$bf_{Student}$: [512 bytes per page / 111 bytes per Student record] = 4 records/page

c) How many pages are needed to store the file?

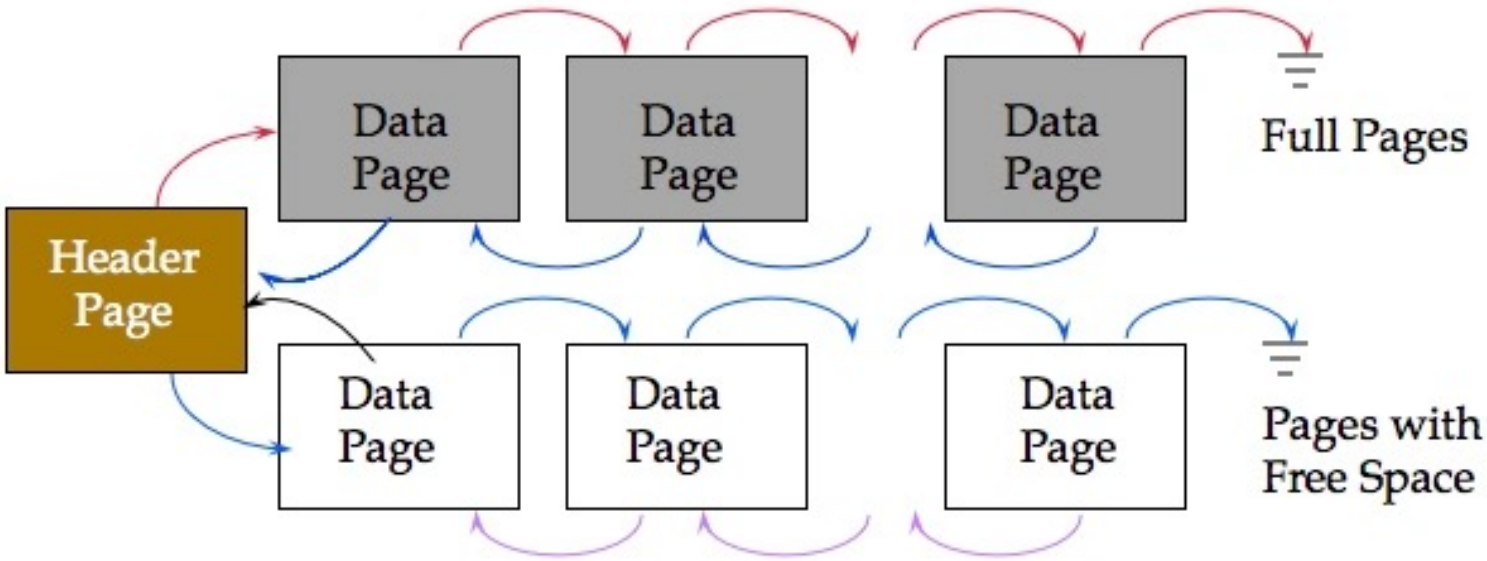
Pages Needed: [20,000 records / 4 records per page] = 5000

Heap File Organization

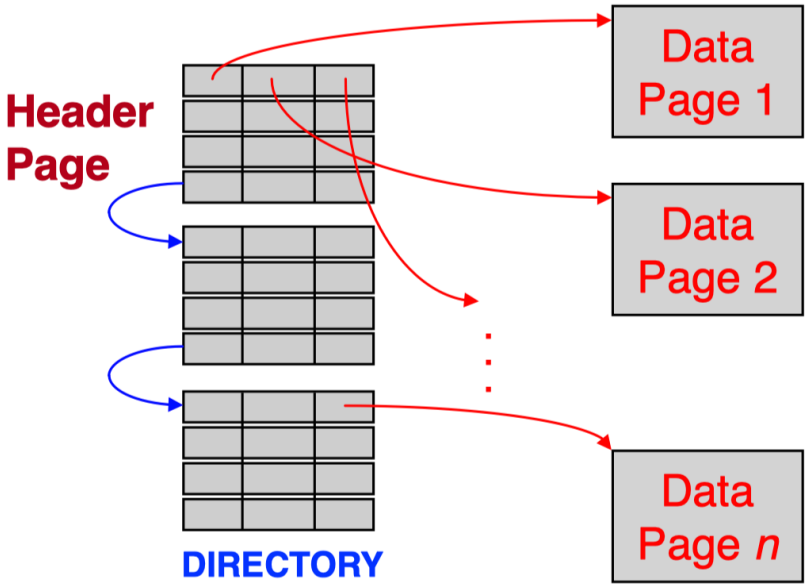
- A record can be placed anywhere in the file where there is space (usually at the end)
  - There is no relationship between a search key and a record's location
  - Once placed, records are usually not moved
  - As a file grows and shrinks, disk pages are allocated/de-allocated
- To support record-level operations, we need to keep track of the:
  1. Pages in a file
  2. Free Space in pages
  3. Records in a page
- Some ways to manage this information:
  - Linked List: a header page points to full and free data pages, which link to each other
  - Page Directory: header pages, which link to each other, point to data pages

Heap File Using Linked List

- Linked List: a header page points to full and free data pages, which link to each other
  - When free page becomes full, move it to the front of full page linked list



Heap File Using a Page Directory



- The entry for a page can include the number of free bytes on the page
  - Free space map: store fraction of space that are free
  - Example: 3 bits to store occupancy

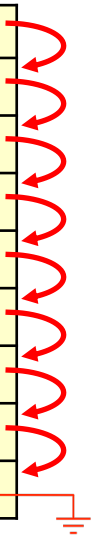
4	2	1	4	7	3	6	5	1	2	0	1	1	0	5	6

- The directory is a collection of pages
  - Linked list implementation is just one alternative

Sequential File Organization

Catalog: account#, branchName, balance

A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	900	
A-222	Redwood	700	
A-305	Round Hill	350	

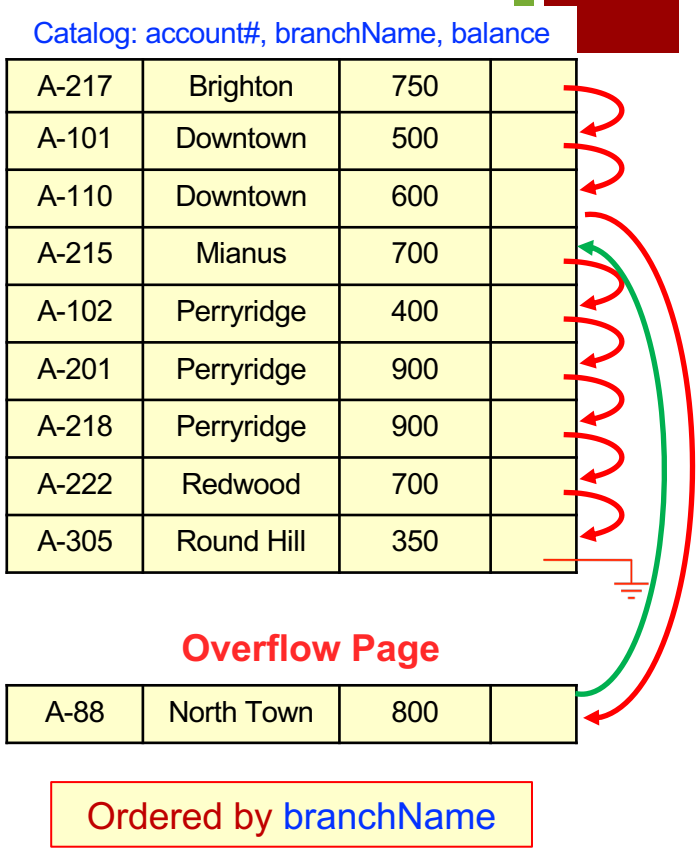


Ordered by branchName

- The records are stored in sequential order, based on the value of a search key (usually, but not always, the primary key)
- Retrieval of records based on the search key is efficient

+ Sequential File Organization

- Insertion – locate the position where the record is to be inserted
  - If there is free space, insert there
  - If no free space, insert the record in an overflow page (can be a heap file)
  - In either case, the pointer chain must be updated
- Deletion – use pointer chains
- Search – binary search on search key;
  - Else sequential file scan
- The file needs to be reorganized periodically to maintain the benefits of the sequential order



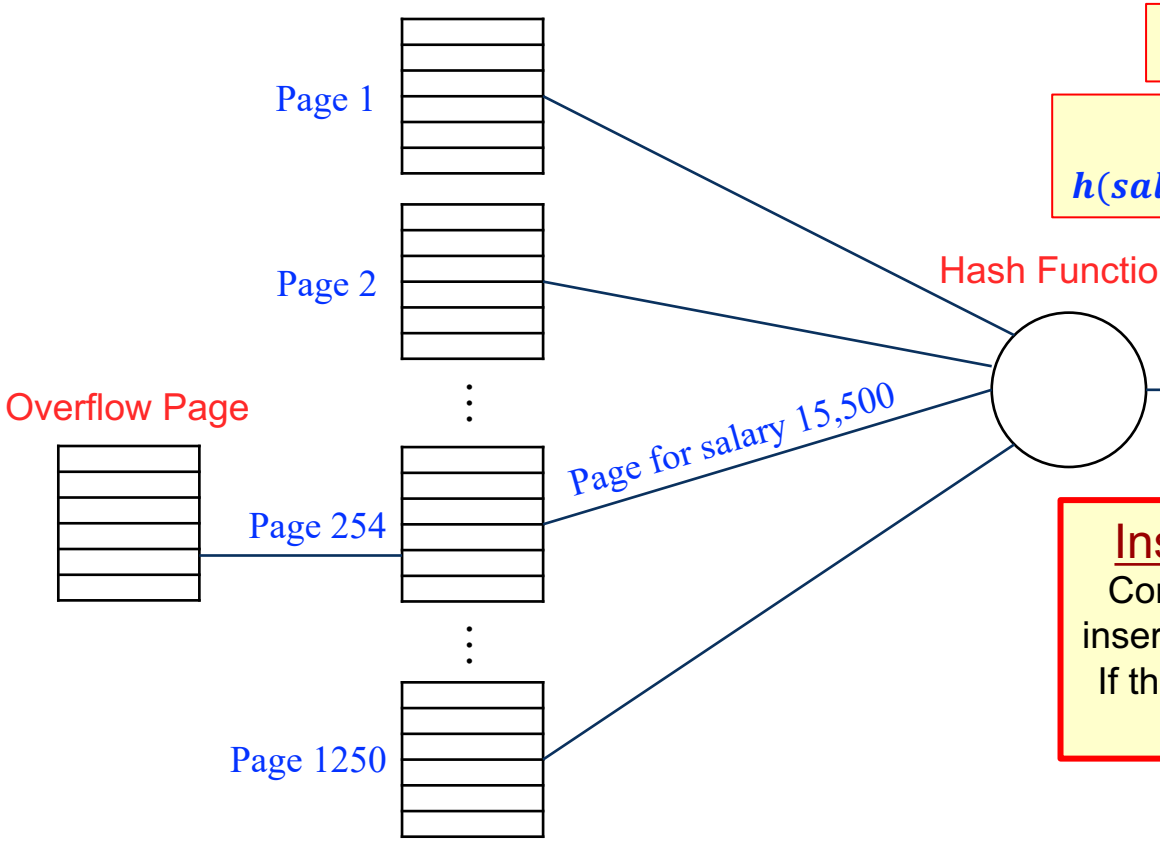
Hash File Organization

- A hash function defines a key-to-address transformation such that a record's physical address can be calculated from its search key value
- The result of a hash function specifies in which page of the file a record should be stored
  - Pages are also referred to as buckets
- Insertion: Apply the hash function to the search key value and store the record in the page calculated by the hash function
  - There is a direct relationship between the search key value and a record's physical location
- Search: Apply the same hash function to the search key value and retrieve the record from the page calculated by the hash function
  - A record can often be retrieved by accessing only one page

Hash File Organization Example

- Assume we want to store an Employee relation with 100,000 records and we can put 100 records per page
  - The blocking factor, $bf_{Employee}$, is 100
- Consequently, we need [# records / $bf_{Employee}$] = [100,000 / 100] = 1,000 pages to store the records
- To allow for future insertions, we can allocate 1,250 pages (buckets) for the relation
  - So that pages are only 80% full
- We want to organize the file so that we can efficiently answer equality selections on salary $\Longrightarrow$ salary is the search key
  - Example query: Find all employee records whose salary is equal to $15,500
- To support this query, the hash function might have the form
  - $h(salary) = (a \times salary + b)\ mod\ 1250$

+ Hash File Organization Example



Search Key: Salary

Hash Function

$$h(salary) = (3 \times salary + 4) \bmod 1250$$

Insert a new Employee Record

Compute the hash function of salary and insert the record in the corresponding page. If the page is full, create an overflow page and insert the new record there.

Query

Find all employee records whose salary is equal to X

Compute the hash function of the salary and find the records in the corresponding page

Good for equality search on hash value only!

Simplistic Analysis of File Organizations

- When analyzing and comparing different file organizations, for simplicity we will:
  - Ignore CPU costs (e.g., searching a page in memory)
  - Approximate I/O costs by ignoring I/Os saved due to page prefetching
  - Do average-case analysis using several simplistic assumptions
    - Single record insert and delete
    - For heap files
      - Equality selection is on the key; therefore, there is exactly one match
      - A record insert is always at the end of a file
    - For sequential files
      - File compaction happens after a record deletion
      - Selection is on sort field(s)
    - For hash files
      - No overflow pages
      - 80% page occupancy

+ Page I/O Cost of Operations



Operation	Heap File	Sequential File	Hash File
Scan all records	B	B	1.25[1] B
Equality Search[2]	0.5B	$\log_2 B$[3]	1
Range Search	B	$\log_2 B$+ # of pages with matches	1.25[1] B
Insert	2[4]	Equality Search + B[5]	2
Delete	Equality Search+1	Equality Search + B[5]	2

B is the number of pages in a file

1. Assumes 80% occupancy of pages to allow for future additions. Thus, 1.25B pages are needed to store all records.
2. Assumes the search is on the key value.
3. Assumes binary search is used.
4. Assumes the record is inserted at the end of the file – read last page and write it back.
5. Assumes insert/delete is in the middle of the file and need to read and write all pages in second half of the file.

+ Exercise 2

Pages Needed: [20,000 records / 4 records per page] = 5000

- A Student file has 20,000 records of fixed-length. Assume the page size is 512 bytes and each record has the following fields
  - name: 30 bytes, studentId: 8 bytes, address: 40 bytes, phone: 8 bytes,
  - birthdate: 8 bytes, gender: 1 byte, majorDeptCode: 4 bytes, minorDeptCode: 4 bytes,
  - classCode: 4 bytes, degreeProgram: 3 bytes.
- How many page I/Os are needed to search for a record given its studentId value if the file of Exercise 1 is organized as

a) A heap file?

Search cost: [5,000 / 2] = 2500 page I/Os (linear search)

b) A sequential file sorted on studentId?

Search cost: $\lceil \log_2 5000 \rceil$ = 13 page I/Os (binary search)

+ Exercise 3

bf = [# bytes per page / # bytes per record]
#pages = [# records / bf_r]

- An Employee file has 30,000 records of fixed-length. Assume the page size is 1000 bytes and each record has the following fields
  - name: 25 bytes, ID: 8 bytes, address: 35 bytes, deptCode: 8 bytes, phone: 8 bytes,
  - birthdate: 8 bytes, gender: 1 byte, jobCode: 3 bytes, salary: 4 bytes,
  - An additional byte is used as a deletion marker

a) What is the record size in bytes?

Record size:25 + 8 + 35 + 8 + 8 + 8 + 1 + 3 + 4 + 1 = 101 bytes

b) What is the blocking factor $bf_{employee}$?

$\boldsymbol{bf_{Employee}}$: [1000 bytes per page / 101 bytes per record] = [9.9] =9 records/page

c) How many pages are needed to store the file?

Pages needed: [30000 records / 9 records per page] =[3333.3] = 3334 pages

+ Exercise 4

- An Employee file has 30,000 records of fixed-length. Assume the page size is 1000 bytes and each record has the following fields
  - name: 25 bytes, ID: 8 bytes, address: 35 bytes, deptCode: 8 bytes, phone: 8 bytes,
  - birthdate: 8 bytes, gender: 1 byte, jobCode: 3 bytes, salary: 4 bytes,
  - An additional byte is used as a deletion marker

For the file of Exercise 3, how many page I/Os are needed to search for

a) A record given its ID value if the file is organized as a sequential file sorted on ID?

Search cost: $\lceil \log_2 3334 \rceil = \lceil 11.7 \rceil = 12$ page I/Os (binary search)

b) If the file is organized as a sequential file sorted by ID, will all the records with a given jobcode value be included?

Search cost: 3334 page I/Os (linear search)

c) A record given its ID value if the file is organized as a hash file hashed on ID and there are no overflow pages?

Search cost: 1 page I/O

+ Outline

1. Physical Storage Media Overview
   - HDD | I/O Cost | RAID levels
2. Database Buffer Manager
3. Record Organization
   - Fixed-length records | Free-lists | Slotted Page
4. **File Organizations**
   - Heap File | Sequential File | Hash File
5. **Column-Oriented Storage Formats**
   - Storage Model | Encoding | Nested Data | Parque | ORC
6. **SIMD-Optimized Advanced Compression**
   - BtrBlocks | FastLanes | BitWeaving

+ Database Workload Comparison

- **OLTP** (Online Transaction Processing) workloads
  - Use *Indexes* to find individual tuples without performing sequential scans
    - For queries with low selectivity predicates
    - Also need to accommodate incremental updates

OLTP Example

```
SELECT balance
FROM accounts
WHERE account_id = 789;
```

- **OLAP** (Online Analytical Processing) workloads
  - Perform *Sequential Scans* on large segments of read-only data
    - Queries access only a small subset of all columns for aggregation
    - Row-based storage waste I/O on unnecessary attributes
    - Columnar storage reconstructs full tuples by combining values from separate column chunks

OLAP Example

```
SELECT region, SUM(revenue), AVG(profit)
FROM sales
WHERE sale_date BETWEEN '2024-01-01' AND '2024-12-31'
GROUP BY region;
```

+ Sequential Scan Optimizations

- Data Encoding / Compression
- Prefetching
- Parallelization
- Clustering / Sorting
- Late Materialization
- Materialized Views / Result Caching
- Data Skipping
- Data Parallelization / Vectorization
- Code Specialization / Compilation

+ Storage Models

- A DBMS's storage model specifies how it physically organizes tuples on disk and in memory

1. **NSM** N-ary Storage Model
   - All the previous discussed methods
   - The DBMS stores (almost) all the attributes for a single tuple contiguously in a single page
   - Ideal for OLTP workloads: point access to individual tuples, frequent inserts and updates
   - Use the tuple-at-a-time iterator processing model
2. **DSM** Decomposition Storage Model
   - Column-store
3. **PAX** Hybrid Storage Model

Column-Stores vs. Row-Stores: How Different Are They Really? SIGMOD'08

DSM Decomposition Storage Model

- The DBMS stores a single attribute for all tuples contiguously in a block of data
  - Classic pure column-store design (MonetDB)
- Ideal for OLAP workloads where read-only queries perform large scans over a subset of the table's attributes
  - Use a batched vectorized processing model
    - Process batches of many column values together, instead of one tuple at a time
    - CPU SIMD instructions
- File sizes are larger (100s of MBs), but it may still organize tuples within the file into smaller groups

DSM: Physical Organization

- Maintain a separate file per attribute, with a dedicated header for column-level metadata
  1. Store Attribute Values in separate fixed-length arrays
     - Most systems identify unique physical tuples using offsets into these arrays
  2. Store Meta Data and Null Bitmap in additional companion arrays
     - Null Bitmap: 1 bit per tuple, marking whether this entry is NULL
       - Works naturally with vectorized processing

	Col A	Col B	Col C
Row 0	a0	b0	c0
Row 1	a1	b1	c1
Row 2	a2	b2	c2
Row 3	a3	b3	c3
Row 4	a4	b4	c4
Row 5	a5	b5	c5

	Meta Data		Null Bitmap				
File 1	a0	a1	a2	a3	a4	a5	…
File 2	b0	b1	b2	b3	b4	b5	…
File 3	c0	c1	c2	c3	c4	c5	…

DSM: Tuple Identification

- How to recover the original tuple?
  - Stitch tuple back together, time-consuming, …

1. Fixed-Length Offsets

- Each value is the same length for an attribute
  - Use simple arithmetic to jump to an offset to find a tuple
- Need to convert variable-length data into fixed-length values
  a) Padding: Simple but space wasteful
  b) Dictionary Compression: Convert repetitive variable-length data into fixed-length values (typically 32-bit integers)

	A	B	C	D
0				HKUST
1				CUHK
2				HKU
3				HKUST

Dictionary

0→ HKUST
1→ CUHK
2→ HKU

D
0
1
2
0

2. Embedded Tuple IDs

- Each value is stored with its tuple ID in a column
- Need auxiliary data structures to find offset within a column for a given tuple ID
  - Space-consuming, slow, seldom used

A		B		C		D	
0		0		0		0	
1		1		1		1	
2		2		2		2	
3		3		3		3	

+ PAX Storage Model





Apache orc™

- Observations
  - OLAP queries almost never access a single column in a table by itself
    - At some point during query execution, the DBMS must get other columns and stitch the original tuple back together
  - But the DBMS needs to store data in a columnar format for storage + execution benefits
  - ➢ We need columnar scheme that still stores attributes separately but keeps the data for each tuple physically close to each other…

- **Partition Attributes Across (PAX)**
  - A hybrid storage model that vertically partitions attributes within a database page
  - ❖ Parquet (Twitter and Cloudera)
  - ❖ ORC Optimized Record Columnar File (Meta/Facebook)

  - ➢ Goal: Obtain the performance benefits of columnar storage
    - While preserving the spatial locality advantages of row storage

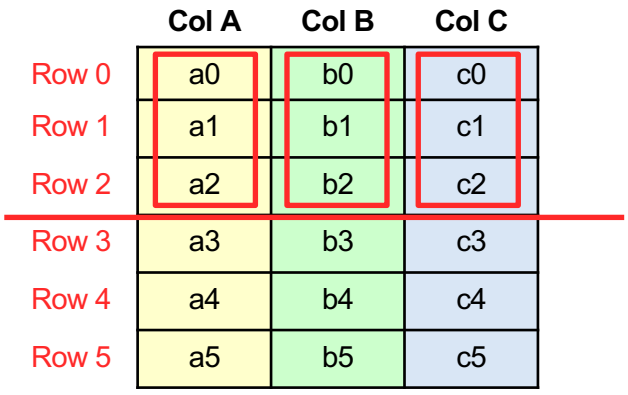
Data page layouts for relational databases on deep memory hierarchies? VLDBJ'02

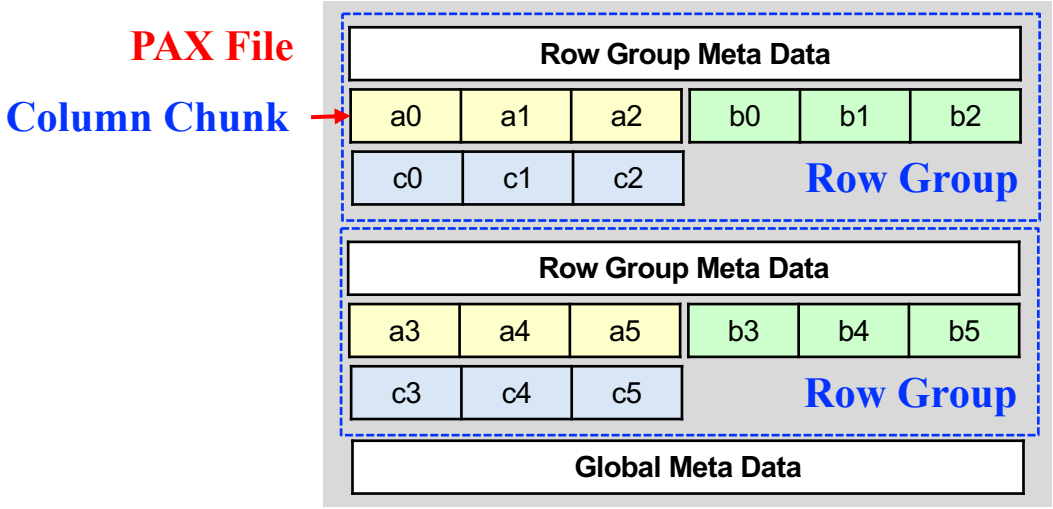
+ PAX Physical Organization

- Organization
  1. Horizontally partition data into *Row Groups*
     - Default 128MB in Parquet
  2. Then vertically partition their attributes into *Column Chunks*
     - Mini-page for one column inside this row group

- Global Meta Data directory contains offsets to the file's row groups
  - This is stored in the footer if the file is immutable
    - Write many row groups one after another to the file
    - While writing, you do not know the byte offset of each row group until you finish writing it
    - After **all row groups are fully written**, you append the global footer at the end of the file

- Each Row Group contains its own meta data header about its contents





+ Open-Source Persistent Data Formats

- Traditional RDBMS: **Tightly coupled**
  - Physical storage format is internal & proprietary to the DBMS
  - Data sharing between system through CSV, JSON, XML, …
  - NSM model optimized for OLTP
- Open Persistent Formats: **Decoupled**
  - Standard file specification, independent of query engines
    - Multiple systems can read the same file
  - Column storage model optimized for OLAP

- HDF5 (1998)
  - Multi-dimensional arrays for scientific workloads
- Apache Avro (2009)
  - Row-oriented format for Hadoop that replace SequenceFiles
- **Apache Parquet** (2013)
  - Compressed columnar storage from Cloudera/Twitter for Impala

- **Apache ORC** (2013)
  - Compressed columnar storage from Meta for Apache Hive
- Apache Carbon Data (2016)
  - Compressed columnar storage with indexes from Huawei
- Apache Arrow
  - In-memory compressed columnar storage from Pandas/Dremio (not persistent)

+ Format Design Decisions

- Every persistent data format makes trade-offs across these dimensions
  - These choices govern I/O cost, compression ratio, query performance, and supported data types.

1. **File Meta-Data**
   - Information about the file, row groups, columns, offsets, statistics (global footer in Parquet)
2. **Format Layout**
   - How to arrange data: row-store (NSM), column-store (DSM), or hybrid (PAX)
3. **Type System**: Data types
4. **Encoding Schemes**
   - Value-level compression: dictionary, RLE, delta, bit-packing
5. **Block Compression**
   - Compress chunks of data (per column chunk / row group)
6. **Filters**: Predicate pushdown helpers: min/max, bloom filters, null bitmaps
7. **Nested Data**
   - Support complex types (struct, list, map) common in modern big data

An Empirical Evaluation of Columnar Storage Formats. VLDB'23
A Deep Dive into Common Open Formats for Analytical DBMSs. VLDB'23

File Meta Data

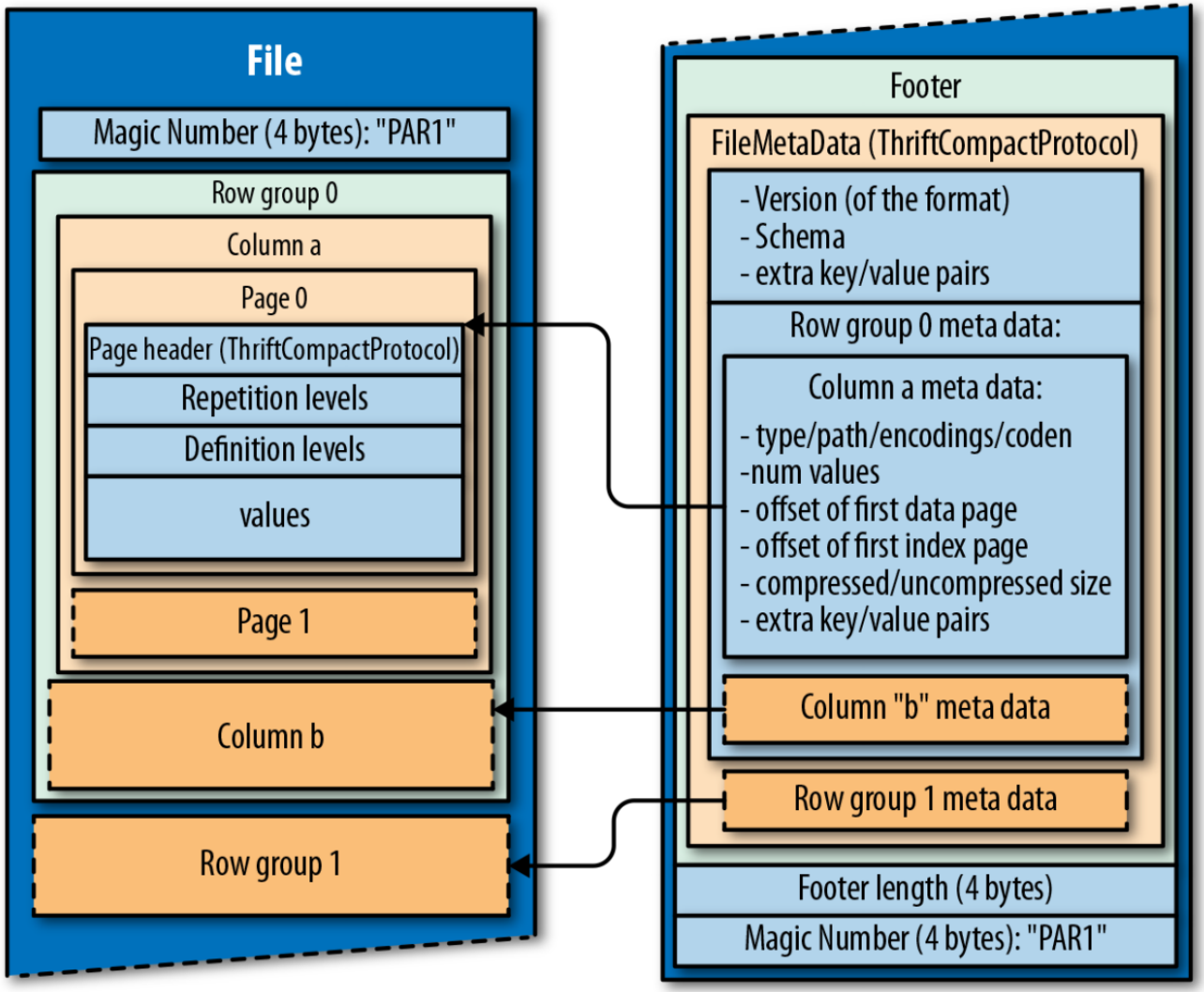
- Files are self-contained to increase portability
  - They contain all the necessary information to interpret their contents without external data dependencies
    - Any DB engine can load and parse the file directly
      - The core idea behind data lake
    - Traditional DB file: coupled with the database system
  - Each file maintains global meta-data (usually in its footer) about its contents:
    1. **Table Schema** (Serialized / Compressed with Thrift, Protobuf)
    2. **Row Group Offsets / Length**
       - Tell the engine where each row group starts and ends inside the file
    3. **Tuple Counts / Zone Maps**
       - Zone Maps: Store min/max statistics for predicate pushdown

+ Format Layout

- The most common formats use the PAX storage model that splits data row groups that contain one or more column chunks
- The size of *Row Groups* varies per implementation and makes compute/memory trade-offs:
  1. **Memory Consumption**
     - The query engine loads partial or full row-groups into memory
     - Larger row-groups increase memory footprint
  2. **Vectorization / Batch Processing Efficiency**
     - Vectorization requires substantial data to realize benefits
     - Tiny row-groups negate vectorization gains via startup overhead
  3. **I/O Efficiency**
     - Row-groups are the base unit for compression and I/O
     - Proper sizing balances random-I/O and sequential-I/O performance

  - Parquet / Arrow: *Number of tuples* (1 million)
    - Bound the row group number regardless of length: Higher vectorization but consumes large memory
  - Orc: *Physical Storage Size* (250 MB)
    - Bound the size, memory constrained, lower vectorization
  - Arrow: *Number of tuples* (1024×1024)

+ Parquet Data Organization



+ Type System

- Defines the data types that the format supports
  1. **Physical Layer**: Low-level byte representation (IEEE-754 for float)
     - How raw bytes are stored on disk
     - Parquet stores DATE or DECIMAL as an int32
  2. **Logical Layer**: Auxiliary types that map to physical types
     - *This int32 is not a normal integer; it represents a date*
- Formats vary in the complexity of their type systems
  - Determine how much upstream producer / consumers need to implement
  - **Parquet**: *Minimal set of physical types*
    - Logical types provide annotations that describe interpretation of primitive type data
    - Pro: Simpler encoder/decoder core
    - Tradeoff: Readers must implement extra logic to interpret logical annotations
  - **ORC**: *More complete set of physical types*
    - Pro: Easier for readers to consume complex types
    - Tradeoff: More types to implement in the codec

+ Parquet vs ORC

Apache Parquet

Documentation / File Format / Types

Types

The types supported by the file format are intended to be as minimal as possible, with a focus on how the types effect on disk storage. For example, 16-bit ints are not explicitly supported in the storage format since they are covered by 32-bit ints with an efficient encoding. This reduces the complexity of implementing readers and writers for the format. The types are:

```
- BOOLEAN: 1 bit boolean
- INT32: 32 bit signed ints
- INT64: 64 bit signed ints
- INT96: 96 bit signed ints
- FLOAT: IEEE 32-bit floating point values
- DOUBLE: IEEE 64-bit floating point values
- BYTE_ARRAY: arbitrarily long byte arrays
- FIXED_LEN_BYTE_ARRAY: fixed length byte arrays
```

Apache ORC

HOME | RELEASES | DOCUMENTATION | TALKS | NEWS | DEVELOP

Types

ORC files are completely self-describing and do not depend on the Hive Metastore or any other external metadata. The file includes all of the type and encoding information for the objects stored in the file. Because the file is self-contained, it does not depend on the user's environment to correctly interpret the file's contents.

ORC provides a rich set of scalar and compound types:

- Integer
  - boolean (1 bit)
  - tinyint (8 bit)
  - smallint (16 bit)
  - int (32 bit)
  - bigint (64 bit)
- Floating point
  - float
  - double
- String types
  - string
  - char
  - varchar
- Binary blobs
  - binary
- Decimal type
  - decimal
- Date/time
  - timestamp
  - timestamp with local time zone
  - date
- Compound types
  - struct
  - list
  - map
  - union

+ Encoding Schemes

- An encoding scheme specifies how the format stores the bytes for contiguous column values
  - Multiple encodings may be stacked together to boost compression

1. **Dictionary Encoding**
   - Map repeated values to small integer IDs
   - Good for low-cardinality columns
   - Most widely used one
2. Run-Length Encoding (RLE)
   - Encode repeated runs of identical values as (count, value)
   - Excellent for long repeated sequences
3. Bitpacking
   - Pack integers into only the bits needed for their maximum value
   - 5-bit integers instead of full 32-bit ints
4. Delta Encoding
   - Store differences between successive values instead of raw values
   - Great for sorted numeric/time series columns
5. Frame-of-Reference (FOR)
   - Store a base value (frame) + small offsets from this base
   - Works well for clustered numeric data

+ Dictionary Compression

- When many rows repeat the same value
  1. Replace frequent values with smaller fixed-length codes
  2. Then maintain a mapping (dictionary) from the codes to the original values
  - Codes could either be positions (using hash table / sorted arrays) or byte offsets into dictionary
  - Optionally sort values in dictionary
    - Binary search to check existence or do range filtering without scanning all values
  - Further compress dictionary and encoded columns
- The compression breakdown when the Number of Distinct Values (NDV) in a column chunk is too large
  - If every value is unique, the dictionary becomes almost as big as the raw data, and codes cannot save space
    - Parquet: Enforces a max dictionary size limit (1 MB)
      - Switches to alternative encoding once exceeded
    - ORC: Pre-compute NDV and disable if too large

+ Dictionary Compression

Original Data

Name
William
Andrea
Andy
Matt
Andy
Andy
Andy
Andy

Unsorted Dictionary

Length	Value	Pos	Offset
6	Andrea	1	0
7	William	0	7
4	Andy	2	13
4	Matt	3	17
		2	13
		2	13
		2	13
		2	13

Sorted Dictionary

Length	Value	Pos	Offset
6	Andrea	3	14
4	Andy	0	0
4	Matt	1	7
7	William	2	11
		1	7
		1	7
		1	7
		1	7

Design Decisions

1. Eligible Data for Dictionary Encoding
   - Parquet: All data types
   - ORC: Only strings
2. Encode the dictionary integer codes (stacked encodings)
   - Parquet: RLE + Bitpacking
   - ORC: RLE, Delta Encoding, Bitpacking, FOR
3. Expose dictionary to query engine for predicate evaluation
   - Parquet: Not supported
   - ORC: Not supported

Procella Unifying serving and analytical data at YouTube. VLDB'19

+ Block Compression

- Compress data using a general-purpose algorithm
  - Operates on a contiguous block of pre-encoded bytes; only sees bytes inside that block
  - Key Considerations
    1. **Computational Overhead**
       - CPU cost for compression/decompression
    2. **Compress vs. Decompress speed Tradeoff**
       1. LZ4 / Snappy: Very fast decompression, modest compression ratio, widely used in analytical DB
       2. Zstd: Tunable; better compression ratio, higher CPU cost
       3. LZO: Legacy, less common now
    3. **Data Opaqueness**
       - Block compressor treats bytes as opaque binary
       - Cannot understand column semantics

+ Compression Algorithms

v·t·e	Data compression methods	[hide]
Lossless type	Entropy	Adaptive coding · Arithmetic · Asymmetric numeral systems · Golomb · Huffman (Adaptive · Canonical · Modified) · Range · Shannon · Shannon–Fano · Shannon–Fano–Elias · Tunstall · Unary · Universal (Exp-Golomb · Fibonacci · Gamma · Levenshtein)
	Dictionary	Byte-pair encoding · Lempel–Ziv (842 · LZ4 · LZJB · **LZO** · LZRW · LZSS · LZW · LZWL · Snappy)
	Other	BWT · CTW · CM · Delta (Incremental) · DMC · DPCM · Grammar (Re-Pair · Sequitur) · LDCT · MTF · PAQ · PPM · RLE
	Hybrid	LZ77 + Huffman (Deflate · LZX · LZS) · LZ77 + ANS (LZFSE) · LZ77 + Huffman + ANS (Zstandard) · LZ77 + Huffman + context (Brotli) · LZSS + Huffman (LHA/LZH) · LZ77 + Range (LZMA · LZHAM) · RLE + BWT + MTF + Huffman (bzip2)
Lossy type	Transform	Discrete cosine transform (DCT · MDCT) · DST · FFT · Wavelet (Daubechies · DWT · SPIHT)
	Predictive	DPCM (ADPCM) · LPC (ACELP · CELP · LAR · LSP · WLPC) · Motion (Compensation · Estimation · Vector) · Psychoacoustic
Audio	Concepts	Bit rate (ABR · CBR · VBR) · Companding · Convolution · Dynamic range · Latency · Nyquist–Shannon theorem · Sampling · Silence compression · Sound quality · Speech coding · Sub-band coding
	Codec parts	A-law · μ-law · DPCM (ADPCM · DM) · FT (FFT) · LPC (ACELP · CELP · LAR · LSP · WLPC) · MDCT · Psychoacoustic model
Image	Concepts	Chroma subsampling · Coding tree unit · Color space · Compression artifact · Image resolution · Macroblock · Pixel · PSNR · Quantization · Standard test image · Texture compression
	Methods	Chain code · DCT · Deflate · Fractal · KLT · LP · RLE · Wavelet (Daubechies · DWT · EZW · SPIHT)
Video	Concepts	Bit rate (ABR · CBR · VBR) · Display resolution · Frame · Frame rate · Frame types · Interlace · Video characteristics · Video quality
	Codec parts	DCT · DPCM · Deblocking filter · Lapped transform · Motion (Compensation · Estimation · Vector) · Wavelet (Daubechies · DWT)
Theory		Compressed data structures (Compressed suffix array · FM-index) · Entropy · Information theory (Timeline) · Kolmogorov complexity · Prefix code · Quantization · Rate–distortion · Redundancy · Symmetry · Smallest grammar problem
Community		Hutter Prize
People		Mark Adler · David A. Huffman · Phil Katz

+ Filters

- Lightweight metadata structures that allow the query engine to skip entire row groups without loading or decompressing data.
  - They are used for predicate pushdown and I/O pruning
  1. **Zone Maps**
     - Maintain min/max values per column at the file-level and row group-level
       - If query predicate falls outside [min, max], skip this row group entirely
     - More effective if values are clustered
     - By default, both Parquet and ORC store zone maps in the header of each row group
  2. **Bloom Filters**
     - A probabilistic structure testing whether a value definitely does NOT exist in a row group
       - No false negatives, but possible false positives
       - Pruning improves when identical values cluster into a small number of row groups
     - Parquet uses Split Block Bloom Filters from Impala

+ Nested Data

- Real-world data sets often contain semi-structured objects
  - JSON, Protobufs, XML, …
  - Flexible Schema: schema evolves easily without rewriting all existing data
  - Preserves natural hierarchical structure of events / logs / messages
  - No costly upfront ETL to flatten nested records into rigid relational tables
  - Hard to support by NSM (Row-based)
    - Raw JSON BLOBL Fully flattened table?
    1. Hierarchical object nesting
    2. Optional / missing fields
    3. Repeating array elements
- Why store semi-structured data in columnar format?
  - Preserve original semi-structured schema (no forced flat table conversion)
  - Selectively read only needed nested subfields (no full object deserialization)
  - Retain columnar compression & filter pruning for individual leaf fields
  1. **Record Shredding (Parquet)**
  2. **Length + Presence Encoding (ORC)**

+ Nested Data: Shredding

- Store leaf paths in nested structure as **separate columns**
  - Because the Json Query's are based one the leaf-path values, no need to parse the entire record
  - But there are repeated arrays and optional parent objects (missing). How to recover?

1. **Definition Level d:** How much of the path exists?
   - Follow the path from the root: +1 for each present **optional** field or **repeated** element; **required**+0
   - Stop at the first missing **optional** field or **repeated** field with no elements
   - A leaf value exists only when d reaches that column's maximum
2. **Repetition Level r:** Where does a new element begin?
   - Number repeated nodes along the path from 1, outermost first
   - $r = 0$: first entry of this column in a new Document.
   - Otherwise, r identifies the repeated level that advances relative to the previous entry in the same column

```
message Document {
  required int64 DocId;
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country;
    }
    optional string Url;
  }
}
```

- Leaf Paths
  1. DocId
  2. Name.Language.Code
  3. Name.Language.Country
  4. Name.Url

```
DocId: 10
Name:
  Language:
    Code: 'en-us'
    Country: 'us'
  Language:
    Code: 'en'
  Url: 'http://A'
Name:
  Url: 'http://B'
Name:
  Language:
    Code: 'en-gb'
    Country: 'gb'
```

```
DocId: 20
Name:
  Url: 'http://C'
```

For Name.Language.Country

- $d = 0$: No Name
- $d = 1$: Has Name, No Language
- $d = 2$: Has Name and Language, no Country
- $d = 3$: Has Name, Language, and Country

Name: Depth 1

Language: Depth 2

- $r = 0$: First one in a new Document
- $r = 1$: Same Document, next Name
- $r = 2$: Same Name, next Language

Nested Data: Shredding

For Name.Language.Country

- $d = 0$: No Name
- $d = 1$: Has Name, No Language
- $d = 2$: Has Name and Language, no Country
- $d = 3$: Has Name, Language, and Country

Name: Depth 1

Language: Depth 2

- $r = 0$: First one in a new Document
- $r = 1$: Same Document, next Name
- $r = 2$: Same Name, next Language

```
message Document {
  required int64 DocId;
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country;
    }
    optional string Url;
  }
}
```

repeated

repeated

repeated

repeated

```
DocId: 10
Name:
  Language:
    Code: 'en-us'
    Country: 'us'
  Language:
    Code: 'en'
  Url: 'http://A'
Name:
  Url: 'http://B'
Name:
  Language:
    Code: 'en-gb'
    Country: 'gb'
```

```
DocId: 20
Name:
  Url: 'http://C'
```

- Leaf Paths
  1. DocId
  2. Name.Language.Code
  3. Name.Language.Country
  4. Name.Url

DocId

Value	r	d
10	0	0
20	0	0

Name.Url

Value	r	d
'http://A	0	2
'http://B	1	2
NULL	1	1
'http://C	0	2

Name.Language.Code

Value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2
NULL	0	1

Name.Language.Country

Value	r	d
us	0	3
NULL	2	2
NULL	1	1
gb	1	3
NULL	0	1

+ Nested Data: Shredding Recovery


```
message Document {
  required int64 DocId;
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country;
    }
    optional string Url;
  }
}
```

DocId: 10
Name:
  Language:
    Code: 'en-us'
    Country: 'us'
  Language:
    Code: 'en'
  Url: 'http://A'
Name:
  Url: 'http://B'
Name:
  Language:
    Code: 'en-gb'
    Country: 'gb'

DocId: 20
Name:
  Url: 'http://C'

DocId

Value	r	d
10	0	0
20	0	0

Name.Url

Value	r	d
'http://A	0	2
'http://B	1	2
NULL	1	1
'http://C	0	2

Name.Language.Code

Value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2
NULL	0	1

Name.Language.Country

Value	r	d
us	0	3
NULL	2	2
NULL	1	1
gb	1	3
NULL	0	1

Query: Name.Language.Country="gb"
Result: The first document

1. Search Name.Language.Country, found (gb, 1, 3)
   - The First document because it only encountered 1 $r = 0$
2. Search DocId
   - Find the first document ID = 10
3. Search Name.Url
   - "http://A": Name 1
   - "http://B": Name 2
   - NULL: Name 3 has no Url
4. Search Name.Language.Code
   - (en-us,0 , 2): Name 1's first language code is en-us
   - (en, 2, 2): Name 1's second language code is en
   - (NULL, 1, 1): Name 2, no language
   - (en-gb, 1, 2): Name 3, language code is en-gb
5. Recover Name.Language.Country
   - (us,0 , 3): Name 1's first language's country is us
   - (NULL, 2, 2): Name 1's second language has no country
   - (NULL, 1, 1): Name 2 has no language
   - (gb, 1, 3): Name 3' first language country is gb

+ Nested Data : Length + Presence

- Store paths in nested structure as separate columns
  - Maintain extra metadata columns at every nested level
  - **length (len):** for repeated groups, how many child entries are in this array
  - **presence (p):** Boolean flag indicating whether this optional node exists

```
message Document {
  required int64 DocId;
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country;
    }
    optional string Url;
  }
}
```

DocId: 10
Name:
  Language:
    Code: 'en-us'
    Country: 'us'
  Language:
    Code: 'en'
  Url: 'http://A'
Name:
  Url: 'http://B'
Name:
  Language:
    Code: 'en-gb'
    Country: 'gb'

DocId: 20
Name:
  Url: 'http://C'

DocId

Value	p
10	true
20	true

Name

len
3
1

Name.Url

Value	p
'http://A	true
'http://B	true
	false
'http://C	true

Name.Language

len
2
0
1
0

Name.Language.Code

Value	p
en-us	true
en	true
en-gb	true

Name.Language.Country

Value	p
us	true
	false
gb	true

Outline

1. Physical Storage Media Overview
   - HDD | I/O Cost | RAID levels
2. Database Buffer Manager
3. Record Organization
   - Fixed-length records | Free-lists | Slotted Page
4. **File Organizations**
   - Heap File | Sequential File | Hash File
5. **Column-Oriented Storage Formats**
   - Storage Model | Encoding | Nested Data | Parquet | ORC
6. **SIMD-Optimized Advanced Compression**
   - BtrBlocks | FastLanes | BitWeaving

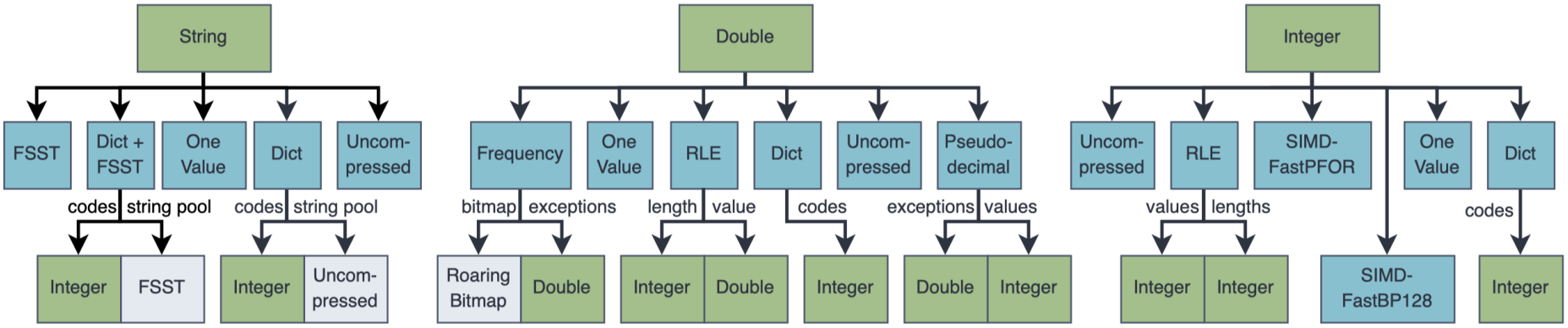
Optimized Columnar Compression for Vectorized Execution

- Critiques of the classic columnar compression
  - Variable-sized Runs
    - Not SIMD friendly
  - Eager Decompression
    - No random access if using block compression
  - Dependencies Between Adjacent Values
    - Due to Delta Encoding and RLE
  - Vectorization Portability
    - ISAs (versions, vendor) have different SIMD capabilities
- Prioritize query execution speed over pure compression ratio
  1. **BtrBlocks**: Adaptive Block Encodings
     - Selects lightweight, SIMD-friendly encodings for individual data blocks
     - Enables vectorized decoding and efficient skipping of irrelevant blocks
  2. **FastLanes**: Fixed SIMD-lane aligned encoding
     - Splits data into fixed-width batches matching SIMD register sizes
     - Eliminates variable-length runs, making SIMD decoding straightforward and parallelizable
  3. **BitWeaving**: Bit-level rearrangement for direct filtering
     - Interleaves identical bit positions across multiple values
     - Allows bitwise filtering operations directly on compressed data without full decompression

+ BTR Blocks

- PAX-based file format with more aggressive nested encoding schemes than Parquet / ORC
- Uses a greedy algorithm to select the best encoding for a column chunk (based on sample) and then recursively tries to encode outputs of that encoding
  - No naïve block compression (Snappy, zstd)
- Store a file's meta-data separately from the data

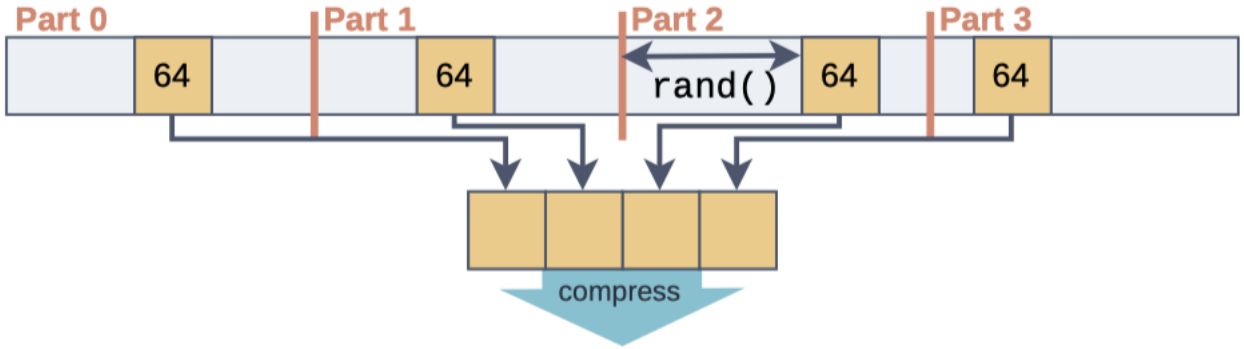
- Encoding Schemes
  - RLE / One Value
  - Frequency Encoding
  - FOR + Bitpacking
  - Dictionary Encoding
  - Pseudodecimals
  - ➢ Fast Static Symbol Table (FSST)
  - ➢ Roaring Bitmaps for NULLs + Exceptions



BtrBlocks: Efficient Columnar Compression for Data Lakes. SIGMOD'23

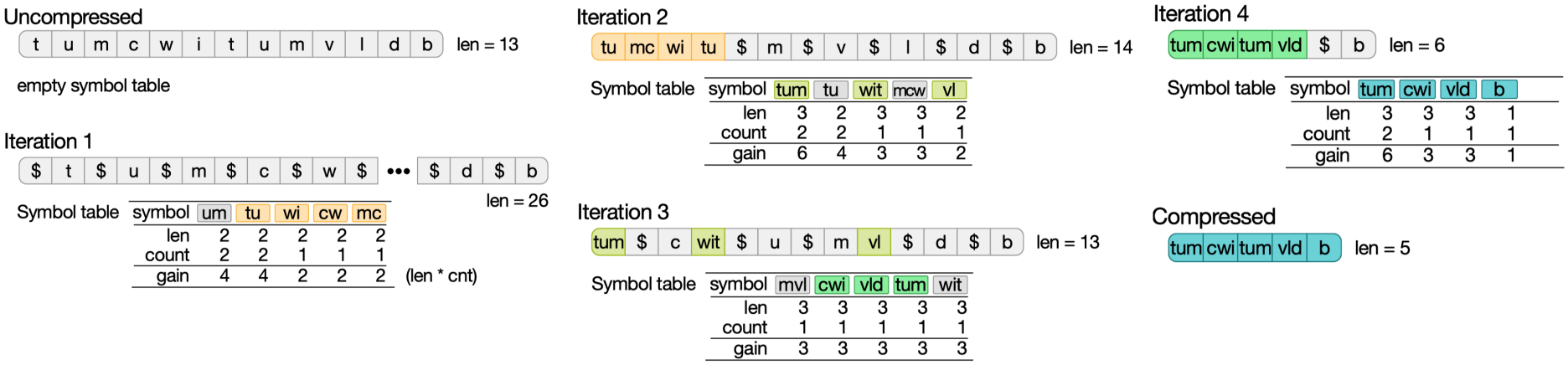
+ BTR Blocks: Encoding Selection

- Collect a sample from the data and then try out all viable encoding schemes.
  - Repeat for three rounds
- Instead of sampling individual values, BtrBlocks selects multiple small runs from non-overlapping random positions
  - For 64k values, it uses 10 runs of 64 values (1% sample size)



+ Fast Static Symbol Table (FSST)

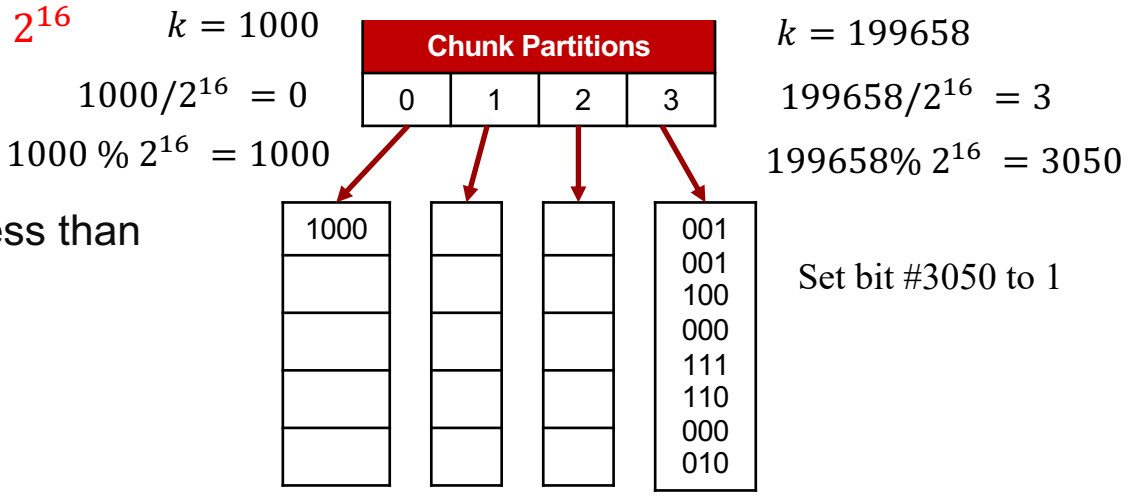
- String encoding scheme that supports random access without decompressing previous entries
  - Replace frequently occurring substrings (up to 8 bytes) with 1-byte codes
  - Uses a "perfect" hash table scheme for fast look-up of symbols without conditionals / loops
    - Construct table using evolutionary algorithm that simply replaces entries on collisions



FSST: Fast Random Access String Compression. VLDB'20

+ Roaring Bitmaps

- Bitmap index that switches which data structure to use for a range of values based local bit density
  - Dense chunks are stored using uncompressed bitmaps
  - Sparse chunks (few 1s for NULL) use bitpacked arrays of 16-bit integers
- Dense chunks can be further compressed with RLE
- For each value k
  - Assign it to chunk $h = k/2^{16}$
  - Position within chunk $l = k \% 2^{16}$
    - If # of values in container is less than 4096, store as array directly
    - Otherwise, store as Bitmap.



Consistently faster and smaller compressed bitmaps with Roaring. Software: Practice and Experience'16

+ BtrBlock Observations

- BtrBlocks + Parquet + ORC generate variable length runs of values
  - This wastes cycles during decoding for both scalar + vectorized operations.
- Parquet + ORC use Delta encoding where each tuple's value depends on the preceding tuple's value
  - This is impractical to process with SIMD because you cannot pass data between lanes in the same register

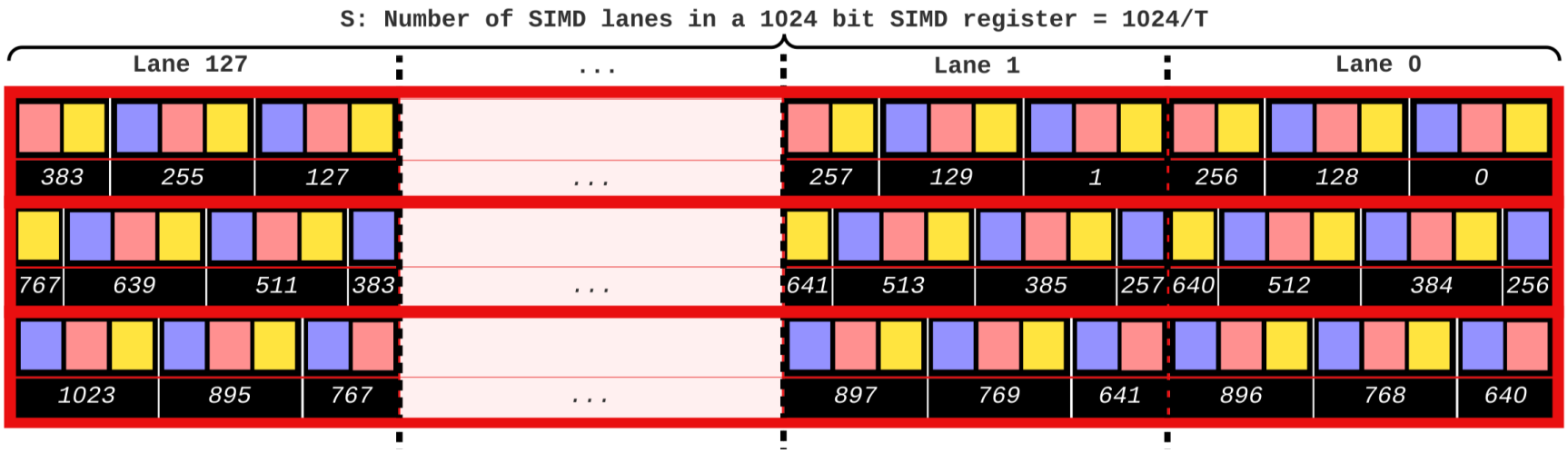
+ FastLanes

- Suite of encoding schemes that achieve better data parallelism through clever reordering of tuples to maximize useful work in SIMD operations
  - Reorganizes data into fixed-size batches, removing variable-length decoding branches that hurt SIMD efficiency
- Similar nested encoding as BtrBlocks:
  - Dictionary
  - FOR
  - Delta
  - RLE
  - Bit-Sliced Encoding
- To future proof format, they define a "virtual" ISA with 1024-bit SIMD registers
- >100 billion integers decoded per second

The FastLanes Compression Layout Decoding > 100 Billion Integers per Second with Scalar Code. VLDB'23

FastLanes: Unified Transposed Layout

- Improve the DBMS's ability to process them efficiently in a vectorized manner using SIMD
  - Reorder values within a column
    - Instead of storing values in original row order one after another, FastLanes shuffles and groups values into fixed-size batches aligned to SIMD lanes
    - This arrangement lets SIMD lanes decode batches independently, eliminating expensive variable-length branches
  - Relational algebra is based on unordered sets, so users should not expect data to be ordered



+ Unified Transposed Layout

Original Data

B	B	C	C	C	C	B	B	B	A	A	A	A	A	A	A
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Run-Length Encoding

Run Values

B	C	B	A
3	2	1	0

Run Lengths

2	4	3	7
3	2	1	0

Delta Encoding

Index Vector

3	3	2	2	2	2	1	1	1	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

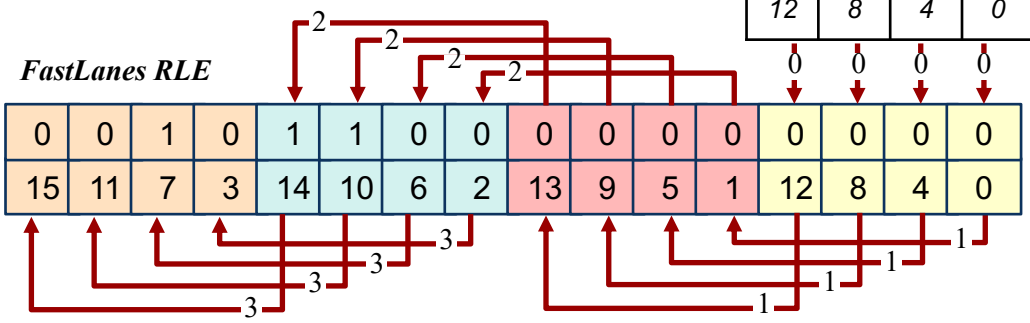
Delta Encoded Vector

0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

2	1	0	0
12	8	4	0

SIMD

FastLanes RLE



Index Vector

Decoded Index Vector

3	3	2	2	2	2	1	1	1	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

+ Observation

- Prior encoding schemes scan data by examining the full value of each attribute
  - All bits of a value at the same time
  - DBMS cannot perform bit-level short-circuit for comparisons on integer types:
    - CPU instructions operate on entire machine word
- What if a DBMS scans only a subset of bits for each value, and checks remaining bits only when necessary?
  - Bit-sliced encoding (bit-plane layout)
    - Group all values bit by bit: store all values' most significant bits together, then the next bit layer, and so on
    - Enables bitwise short-circuit comparison: evaluate higher bit planes first
    - Skip decoding lower bit planes when higher bits already determine the comparison result

+ Bit-Sliced Encoding for Filtering

Original Data | *Bit-Slices*

id	zipcode	null	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	21042	0	0	0	1	0	1	0	0	1	0	0	0	1	1	0	0	1	0
2	15217	0	0	0	0	1	1	1	0	1	1	0	1	1	1	0	0	0	1
3	02903	0	0	0	0	0	0	1	0	1	1	0	1	0	1	0	1	1	1
4	90220	0	1	0	1	1	0	0	0	0	0	0	1	1	0	1	1	0	0
6	14623	0	0	0	0	1	1	1	0	0	1	0	0	0	1	1	1	1	1
7	53703	0	0	1	1	0	1	0	0	0	1	1	1	0	0	0	1	1	1

bin(21042)→ 00101001000110010

```
SELECT *
FROM customer_dim
WHERE zipcode < 15217
```

- Walk each slice and construct a result bitmap
- Skip entries that have 1 in first 3 slices (16, 15, 14)

0	0	0	1	1	1	0	1	1	0	1	1	1	0	0	0	1

+ Bit-Sliced Encoding for Aggregation

- Bit-slices can also be used for efficient aggregate computations
  - SUM(attr) using Hamming Weight (the number of 1s in a bit-slice)
    1. First, count the number of 1s in slice 17 and multiply the count by 2^{17}
    2. Then, count the number of 1s in slice 16 and multiply the count by 2^{16}
    3. Repeat for the rest of slices…
- Use the CPU POPCNT (Population Count) instruction to efficiently count the number of bits set to 1 in a register

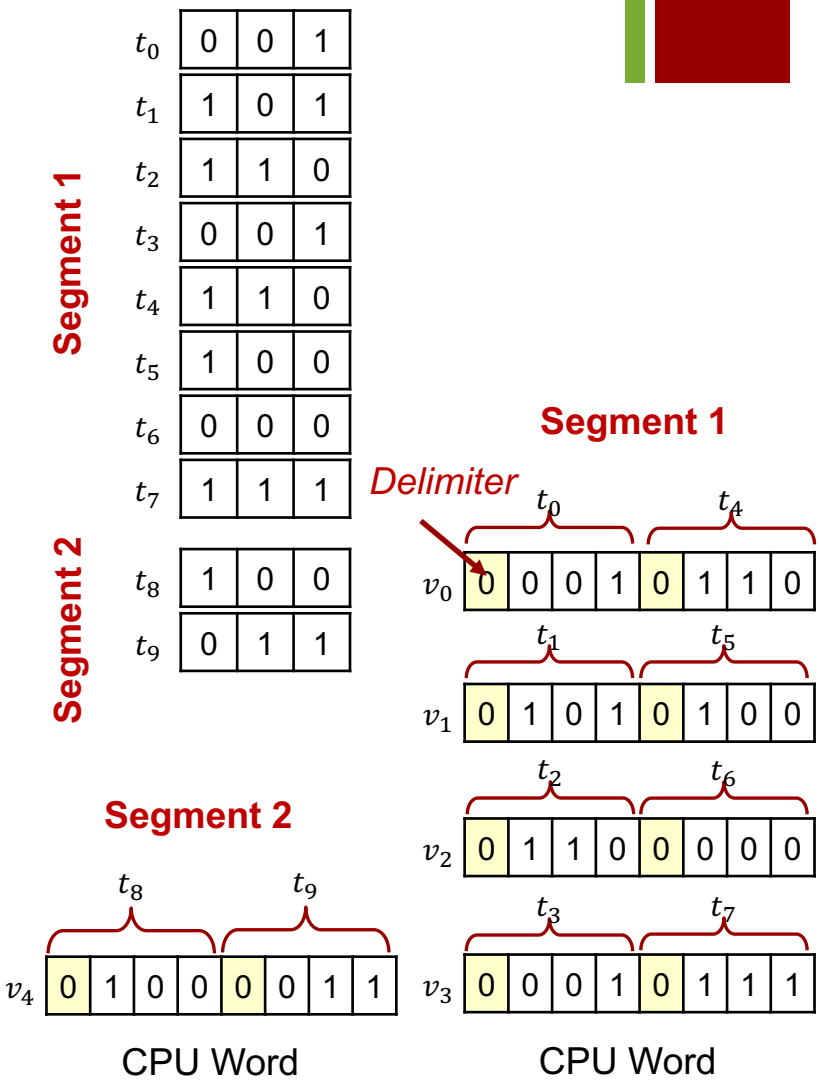
+ BitWeaving

- Alternative encoding scheme for columnar databases
  - It enables efficient predicate evaluation directly on compressed data using SIMD
    - Uses order-preserving dictionary encoding
    - Supports bit-level parallelization
    - Relies only on common CPU instructions
      - No scatter/gather required
- Storage Layout
  1. Horizontal
     - Row-oriented storage at the bit-level
  2. Vertical
     - Column-oriented storage at the bit-level
     - Similar to Bit-Slicing but with SIMD support

BitWeaving: fast scans for main memory data processing. SIGMOD'13

+ BitWeaving Horizontal Storage

- Bit-level row-oriented layout
  - Bits of each individual value are kept together
- Pack multiple compressed values into one CPU processor word
  - A delimiter bit marks the start of each value
  - Values from the same segment are interleaved and packed into machine words
  - Allows SIMD predicate evaluation directly on these packed words, without full decompression



+ BitWeaving Horizontal Query

Segment 1	t_0	0	0	1
	t_1	1	0	1
	t_2	1	1	0
	t_3	0	0	1
	t_4	1	1	0
	t_5	1	0	0
	t_6	0	0	0
	t_7	1	1	1
Segment 2	t_8	1	0	0
	t_9	0	1	1

```
SELECT *
FROM table
WHERE val < 5
```

1 0 1

$X =$ 0 0 0 1 | 0 1 1 0 (t_0, t_4)

$Y =$ 0 1 0 1 | 0 1 0 1 (5, 5)

$mask =$ 0 1 1 1 | 0 1 1 1

Bit-level Testing $X < Y$ $\quad (Y + (X \oplus mask)) \wedge \neg\, mask =$ 1 0 0 0 | 0 0 0 0

$Y + (7 - X) = 7 + (Y - X)$ $\quad 1 < 5 \quad 5 < 6$

Y larger, increase by 1 $\quad$ Y smaller

- Only requires three instructions to evaluate a single word
- Works on any word size and encoding length
- Paper contains algorithms for other operators

Selection Vector

t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7

1 0 0 1 | 0 1 1 0

v_0 0 0 0 1 | 0 1 1 0 (t_0, t_4) $\xrightarrow{<5}$ 1 0 0 0 | 0 0 0 0 $\xrightarrow{\gg 0}$ 1 0 0 0 | 0 0 0 0

v_1 0 1 0 1 | 0 1 0 0 (t_1, t_5) $\xrightarrow{<5}$ 0 0 0 0 | 1 0 0 0 $\xrightarrow{\gg 1}$ 0 0 0 0 | 0 1 0 0

v_2 0 1 1 0 | 0 0 0 0 (t_2, t_6) $\xrightarrow{<5}$ 0 0 0 0 | 1 0 0 0 $\xrightarrow{\gg 2}$ 0 0 0 0 | 1 0 1 0

v_3 0 0 0 1 | 0 1 1 1 (t_3, t_7) $\xrightarrow{<5}$ 1 0 0 0 | 0 0 0 0 $\xrightarrow{\gg 3}$ 1 0 0 1 | 0 0 0 0

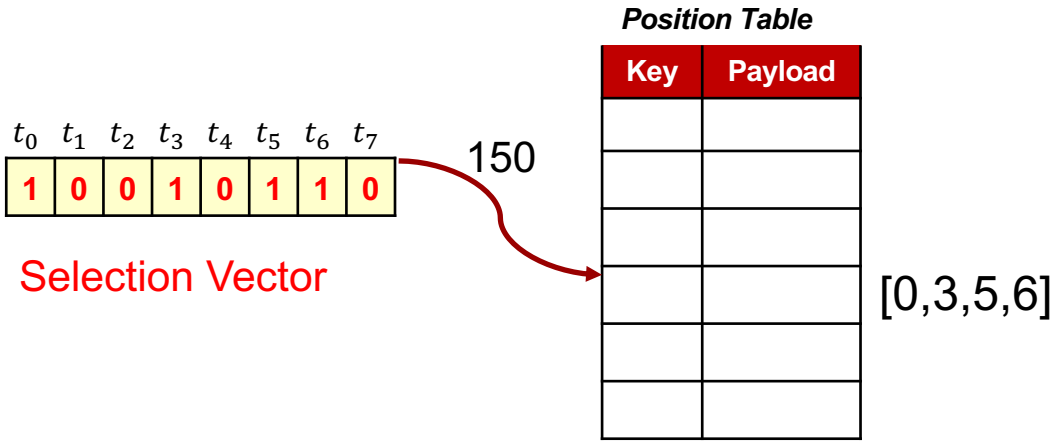
+ Selection Vector

- SIMD Comparison Operators generate a bitmask (selection vector), marking which tuples satisfy the predicate
  - DBMS must convert it into column offsets for downstream operators

- Approach #1 Iteration

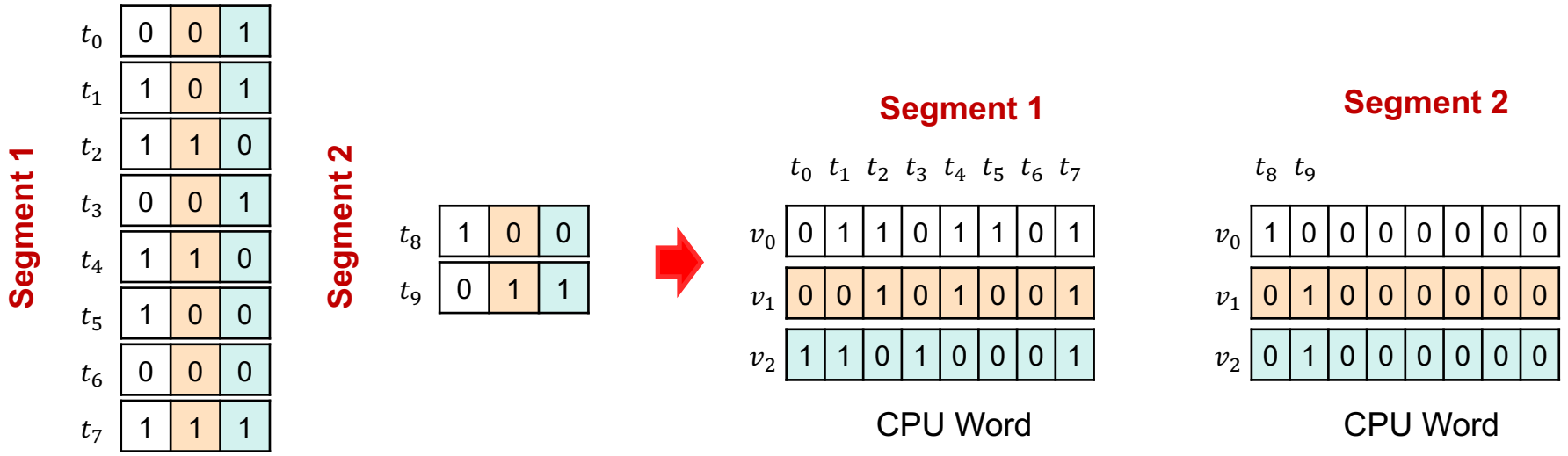
```
tuples = [ ]
for (i=0; i<n; i++) {
    if sv[i] == 1
        tuples.add(i);
}
```

- Approach #2 Per-Computed Position Table

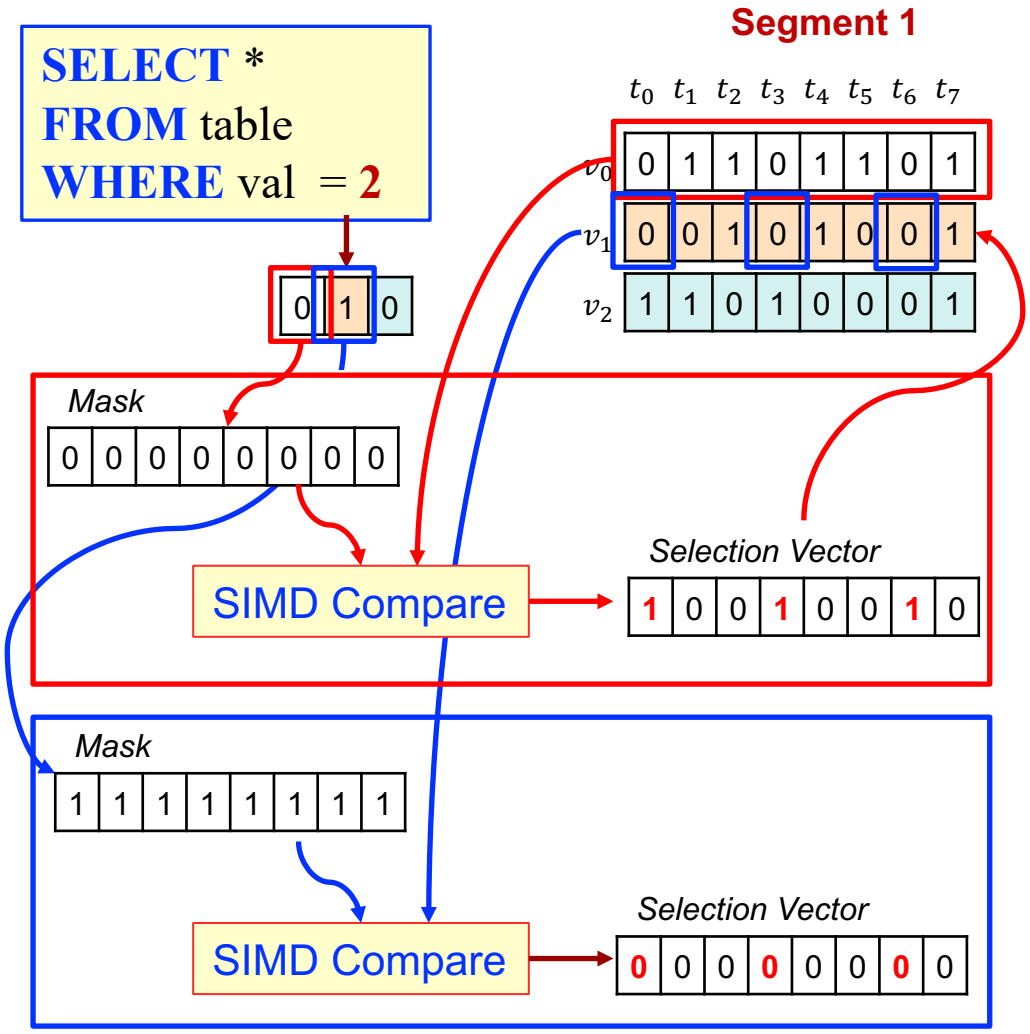


+ BitWeaving Vertical Storage

- Vertical storage (columnar storage / DSM)
  - Store values of the same attribute together, instead of packing all fields of one tuple.
- Data is grouped into segments
  - Each segment contains a batch of tuples, split into separate attribute vectors.
  - Each attribute vector packs values for many tuples into a contiguous processor word (vector register-sized block), optimized for SIMD execution.
  - Queries only load columns they need, reducing I/O and memory traffic, especially for OLAP projection/filter workloads.
- Tradeoff: reconstructing full tuples requires assembling values from multiple separate column vectors



+ BitWeaving Vertical Storage Query



- DBMS can perform early pruning like Bit-Slicing
- Skip the last vector because all bits in previous comparison are zero

+ Summary

- **Storage design determines the cost of accessing and processing data**
  - The same logical query can incur very different I/O, memory traffic, and CPU work under different physical layouts
- **Each storage layer addresses a different problem**
  1. Storage media and RAID balance latency, capacity, and fault tolerance
  2. Buffer management reduces repeated I/O and controls page writeback
  3. Record layouts and heap, sequential, or hash files determine how records are located and updated
- **Analytical storage organizes data for selective reading and efficient computation**
  - NSM, DSM, and PAX provide different locality trade-offs.
  - Parquet and ORC combine columnar organization, metadata, compression, and nested-data representation
  - BtrBlocks, FastLanes, and BitWeaving improve compression, parallel decoding, or filtering on encoded data
- **Choose layouts according to the workload and its bottleneck**
  - Consider predicate selectivity, accessed columns, update frequency, and available memory
  - Evaluate total query cost: bytes avoided and transferred, decoding work, and tuple reconstruction
  - Better compression does not automatically mean faster queries