File Organization and Indexing

DBMS Internals

Two topics:
- file organization and indexing
- query processing and optimization

Our motivation is twofold –
- understanding these topics provides a deeper understanding of database performance and tuning options
  - what indexes are available
  - how you write the query
  - providing hints to the optimizer
  - whether the optimizer is using the most current info
- it's interesting to know what is happening behind the scenes

File Organization

Basic logical unit of data storage is the *record*.
- i.e. one row of a table

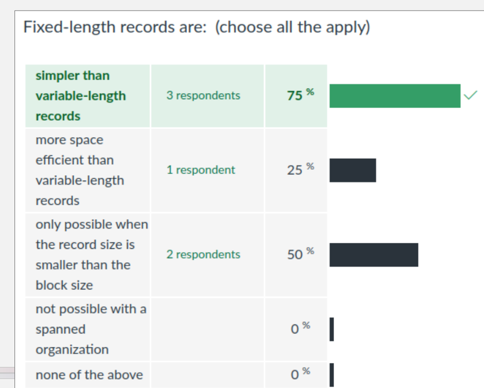
All records in a file are typically of the same type.
- i.e. one file per table

Basic unit of data transfer between disk and memory is the *block*.
- block size is a physical property of the disk

- record size *R* = number of bytes per record
- block size *B* = number of bytes per disk block
- blocking factor *bfr* = number of records per disk block

File Organization

- records can be *fixed*- or *variable*-length
  - variable-length stems from variable-length fields such as VARCHAR



fixed-length fields are more space-efficient that variable-length fields if all of the data values are the same length

whether all records are the same length is independent of how they fit into blocks on the disk

File Organization

- files can have a *spanned* or *unspanned* organization
  - spanned allows records to be split between blocks

An unspanned organization is: (choose all that apply)

simpler than a spanned organization ✓	3 respondents	75 %
more space effcient than a spanned organization		0 %
only possible when the record size is smaller than the block size ✓	2 respondents	50 %
not possible with variable-length records	1 respondent	25 %
none of the above		0 %

how records fit into blocks on the disk is independent of whether or not they are all the same length

Example

R = record size
B = block size
bfr = blocking factor

table	field	type	size (bytes)	R record size (bytes)	average record size (bytes)	average blocking factor (*) unspanned bfr = ⌊B/R⌋
SAILOR	sid	SMALLINT	2			
	sname	VARCHAR(45)	1 – 46			
	rating	TINYINT	1			
	age	DECIMAL(3,1)	2			
BOAT	bid	SMALLINT	2			
	bname	VARCHAR(45)	1 – 46			
	color	VARCHAR(45)	1 – 46			
RESERVATION	sid	SMALLINT	2			
	bid	SMALLINT	2			
	day	DATE	3			

(*) with block size of 1024 bytes

Example

R = record size
B = block size
bfr = blocking factor

table	field	type	size (bytes)	record size (bytes)	average record size (bytes)	average blocking factor (*) unspanned bfr = ⌊B/R⌋
SAILOR	sid	SMALLINT	2	6 – 51	28	1024/28, round down = 36
	sname	VARCHAR(45)	1 – 46			
	rating	TINYINT	1			
	age	DECIMAL(3,1)	2			
BOAT	bid	SMALLINT	2	4 – 94	49	1024/49, round down = 20
	bname	VARCHAR(45)	1 – 46			
	color	VARCHAR(45)	1 – 46			
RESERVATION	sid	SMALLINT	2	7	7	1024/7, round down = 146
	bid	SMALLINT	2			
	day	DATE	3			

(*) with block size of 1024 bytes

File Operations

for traditional HDDs –

- *seek* is the time to position the read/write head for the correct track
- *latency* is the time for the disk to spin the correct block under the head
- *block transfer time* is the time to actually transfer the block from disk to memory
- reading consecutive blocks is much faster than non-consecutive blocks

compared to HDDs, for solid state drives (SSDs) –

- seek time is eliminated
- latency is greatly reduced
- there is little difference between reading consecutive and non-consecutive blocks

When considering the time it takes to carry out a database operation (such as SELECT, INSERT, DELETE, or UPDATE), we focus on the number of disk blocks read rather than the actual elapsed time because: (choose all that apply)

it takes the same amount of time to access any block, so it amounts to the same thing		0 %
the seek and latency times are a physical property of the disk - constant for one system but different from one system to the next ✓	3 respondents	75 %
while reading consecutive blocks may be faster than reading non-consecutive blocks, how the blocks are arranged on disk is controlled by the operating system ✓	4 respondents	100 %

File Operations

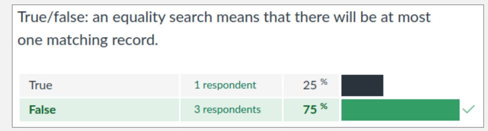
Searching is the most important operation.

- common to have more data retrieval queries than insert/update/delete
- update/delete requires first locating the record(s) involved
- insert often involves some kind of search
  - e.g. to locate insertion point
  - e.g. to verify that key constraints are satisfied

Searching

Types of searches.

- equality comparison (=)
  - at most one match for primary key and unique columns
  - may be multiple matches otherwise



- range comparison (<, >)
- complex conditions
  - equality or range conditions combined with AND, OR

Physical File Organization

b = # blocks
bfr = blocking factor
s = # records matched

Unordered file – records are placed in the file in the order they are inserted.

operation	how done?	blocks accessed
searching (single match)	linear search (stop when found)	max b average b/2
range searching (or multiple matches)	linear search	b
insert (one record)	read last block, add record, write block	2
delete	search + write deletion marker	search + s
update	fixed length: search, then change variable length: delete, then insert	search + s search + 3s

Physical File Organization

b = # blocks
bfr = blocking factor
s = # records matched

Sorted file – records are ordered based on the value of one or more attributes.

operation	how done?	blocks accessed
searching on ordering field (single match)	binary search	ceil($\log_2$(b))
range searching on ordering field	binary search to locate one end of range + scan	ceil($\log_2$(b)) + ceil(s/bfr)
searching / range searching on other field	linear search	max b average b/2 (equality search, single match)
insert (one record)	search + read block, add record, write block (shifting can be mitigated by leaving empty spaces and periodically redistributing space, at the cost of more blocks)	search + 1
delete	search + write deletion marker	search + ceil(s/bfr)
update (one record)	fixed length: search, then change variable length: delete, then insert	search + 1 2*search + 2

Physical File Organization

b = # blocks
bfr = blocking factor
s = # records matched

Hash file – apply hash function to hash field to find address of disk block containing the record.

- consecutive key values generally don't hash to consecutive locations

operation	how done?	blocks accessed
searching on hash field (single match)	hash field value, read that block	1
range searching on hash field	small number of values in range: for each value in the range, hash and read that block	# values checked
	large number of values in range: linear search of the file	b
searching / range searching on other field	linear search	max b average b/2 (equality search, single match)
insert / delete / update (fixed length, one record)	search time + read, modify, write block (if modifying hash field, delete + insert) (shifting can be mitigated with deletion markers and empty spaces, at the cost of more blocks)	search + 1 (2*search + 2)

Physical File Organization

b = # blocks
bfr = blocking factor
s = # records matched

operation		blocks accessed		
		unordered	sorted	hash file
search (single match)		max b average b/2	$ceil(log_2(b))$ (on ordering field)	1 (on hash field)
range search (or multiple matches)		b	$ceil(log_2(b))$ + ceil(s/bfr)	# values checked (few values) b (many values)
insert (one record)		2	search + 1	search + 1 2*search + 2 (if modifying hash field)
delete (one record)		search + 1	search + 1	
update (one record)	fixed length	search + 1	search + 1	
	variable length	search + 3	2*search + 2	

There's a big difference in search time, which underlies everything.

- hash file is best for equality search, but worst or close to worst for range search
- ordered file is good for searching (equality and range)

...but a file can only be ordered/hashed on one field (or group of fields)

Searching

r = # records
d = # distinct values
s = # records matched

Determining the expected number of matches *s* –

- equality comparison (=), primary key and unique columns
  - at most one match
- otherwise assume equal distribution of records unless there is additional information
  - equality comparison (=) – what's the expected number of records per value?
    - if there are *d* unique values – *r/d*
  - range comparison (<, >) – what fraction of the range matches the condition, and how many records is that?