sequence of n nums, stored in N pages on disk

Ex: $n = 24$, $N = 12$ each page stores 2 numbers.

memory capacity is B pages ($B < N$) ($B \geq 3$)

External Sorting

* Divide & conquer splits data set into separate runs and then sorts them individually.

phase 1: Sort blocks of data that fit in dram, write them back in disk

phase 2: combine to a single file. (multiple passes might be needed.)

run: refer to a set of blocks where the nums are already sorted.

$\lceil N/B \rceil$ runs → preliminary pass done.

If B is bigger, can do merge B-1 runs and 1 memory block as output buffer.

- merge by moving smallest number in the input buffers to the output buffer.

1.1 and 1.2 done → finished a merge pass.

Perform another merge pass ⇒ decrease # of runs by factor of B-1

If there are at least B-1 runs left, another merge pass is launched, repeated until number of runs is 1.

pass #0:
use B buffer pages
$\lceil N/B \rceil$ sorted runs of B pages

pass #1, 2, ... :
merge (B-1) runs
(B-1) ways to merge

total num of passes: $1 + \lceil \log_{(B-1)} \lceil N/B \rceil \rceil$

total I/O cost: $2N \cdot$ (# of passes)

N = num of pages, B = num of buffer pages

Ex: $N = 108$, $B = 5$

Pass 0: $\lceil 108/5 \rceil = 22$ runs of 5 pages

Pass 1: $\lceil 22/4 \rceil = 6$ runs of (5×4 pages)

Pass 2: $\lceil 6/4 \rceil = 2$ runs of ($20 \cdot 4$ pages) $= 40$, $108 - 80 = 28$ ← one with

Pass 3: $\lceil 2/4 \rceil = 1$ ← sorted file

Algorithm: Join → join 2 tables

"Intersection" simpler version of join.

M pages in table R, m tuples in R

N pages in table S, n tuples in S

Nested Loop Join:

for each tuple r ∈ R ← outer
  for each tuple s ∈ S ← inner
    emit, if r.match(s)

I/O cost: $M + mN$

M → pages of R   N pages of S
m → tuples   n tuples

NLJ: reads many duplicated data.
  checks entire S block m times yuck.

Block Nested Loop Join:

for each block Br ∈ R:
  for each block Bs ∈ S:
    for each tuple r ∈ Br:
      for each tuple s ∈ Bs:
        emit if r.match(s)

cost: $M + (M \cdot N)$

M pages of outer R
N pages of inner S
smaller table for # pages should be outer

For B memory blocks:

for each B-2 blocks ∈ R:
  for each block Bs ∈ S:
    for each tuple r ∈ Br:
      for each tuple s ∈ Bs:
        emit if r.match(s)

B-2 buffers for scanning R
cost: $M + (\lceil M / (B-2) \rceil \cdot N)$

simple, easy to implement
no need of indexes
needs to sequentially scan entire inner table

Index Nested Loop Join
  built index for inner table S

for each tuple r ∈ R
  for each tuple s ∈ Index($r_i = s_j$)
    emit if r.match(s)

cost: $M + (m \cdot c)$

M pages of R

m tuples of m

C cost of constant

Sort-merge Join → when R and S are already sorted.

Sort cost (R): $2M \cdot (1 + \lceil \log_{B-1} \lceil M/B \rceil \rceil)$ merge cost: $(M+N)$

sort cost (S): $2N \cdot (1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil)$ If already sorted ↑

Hash-join: when join condition is equality

build hash table HT_R for R } build phase, IO cost: M pages of reads for R

for each tuple $s \in S$
output, if $h_1(s) \in HT_R$ } probe phase, IO cost: N pages of reads for S

total cost: $M + N$

Grace-hash join: hash table does not fit in memory.

Partition $R = R_1, \ldots$
$S = S_1, \ldots$

only R_i can be joined with S_i

cost of hash joins: $3(M+N)$

partition phase: $2(M+N)$

probing phase: $M+N$

Transaction: execution of a sequence of one or more operations on a DB to perform some higher-level function

Basic unit of change in DBMS, either all or none, no partial.

properties:
- A Atomicity:
- C consistency:
- I Isolation:
- D Durability:

BEGIN // begin

COMMIT ← stop here or save all changes

ABORT/ROLLBACK ← changes are undone

DBMS → concerned only about read/write data

DB → {A, B, C, ...}

BEGIN:
read (A)
A = A - 100
write (A)
COMMIT:

A: trans. unit of operation executed whole or none
not intermediate results

C: const. state, satisfy constraints

I: each txn executes as if it is executing alone. (isolate from other txns)

D: If commit, results persistent regardless of failures DB can encounter.

"all or nothing", "look correct", "as if alone", "survive failures"

Break acid: concurrent executions, system crashed in the middle
concurrency control, crash recovery

Lost update: read before an update

non-repeatable read: shows available there's none

Dirty read: update → read
rollback ↙
→ reads (temporary/dirty data)

Lock: mechanism to control concurrent access to a data item.

exclusive (X) mode: read/write, shared (S) mode: read-only

Tsx granted if requested lock is compatible with locks on item.

	S	X
S	T	F
X	F	F

any # of tsx can hold shared locks on item.
only one tsx can hold an x lock

xlock → before writing data
slock → before reading data
} avoids lost update, dirty read but not repeatable (ok for single item)

Two-phase Locking

Expanding: need locks, no locks released
Shrinking: release locks, no locks needed

cascade abort: x lock on t released before commit may be locked by another t. (u)
if t aborts ⇒ u aborts

Rigorous 2PL: hold the lock until the end of the transaction.

Concurrent breaks: C, I

crash: A, D

Tsx Failures → logical error: Tsx can't be execute be of internal error.
Internal state error: DBMS must terminate active tsx due to an error condition.

sys. Failures

storage. – –

Software Failure → div by 0

Hardware Failure → computer crashes

Storage-media failure: No DBMS can recover.
restore from archived.

tsx → write to buffer → write to disk

read(A) → A = A ~~ → write(A) → output(A)

WAL: before writing to disk write log to disk first.

Log: changes made, stored in append-only log file / separated from data file.

↓

smaller than actual data.

tuple → log: attributes

page → log: byte

<START T>

<COMMIT T>

<ABORT T>

UNDO Log: has old data <T, X, V>

↑ tsx, ↑ data item, ↑ former value

REDO Log: has new data <T, X, V>

↑ new value

UNDO only: write <T, X, V> to disk before altering X. <COMMIT T> is written only after all dB changes by T commits, tsx have been written to disk

If <COMMIT> is not in log use UNDO to roll back.

must flush all data changed by end of tsxn. (slow)

REDO ONLY:

1) log records indicating changes
2) COMMIT log record
3) the changed data item themselves.

Need to keep all changed data in memory before the tsx is committed. consume a lot of memory space.

UNDO/REDO Log record <T, X, V1, V2> flush all logs before commit

↓ old ↓ new

<commit> → redo tsx with new data

no <commit> → undo tsx with old data

1) log records indicating change
2) commit log record ⤸ flush

No need to flush all data by end of tsx

flush data before/after commit

high performance / low memory consumption.

Flush dirty data before tsx commited → affect A

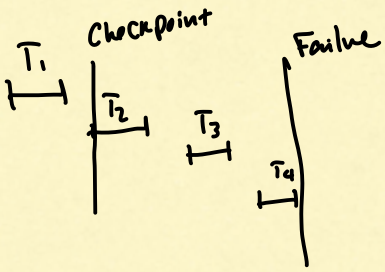
WAL → enforces A

don't flush all data before tsxn is committed → affect D

Checkpoint:

Output on disk all logs in DRAM

Output to disk all modified blocks (dirty pages)



T1 ✓

Redo: T2 and T3

undo: T4

soln: stall tsxns

1) Do not acept new tsxn
2) wait until all tsxn finish
3) flush all log records to disk
4) flush all dirty pages to disk
5) write checkp record on disk
6) resume tsnx processing.

Distributed DB: collection of multiple, logically irrelated DBs distributed over a computer network. HADOOP, SPARK

Scale-up: increase resources

scale-out: increas number of resources

DBMSs → specify shared resources w/ CPU

Shared Nothing: Each DB has own CPU, mem, disk
- nodes communicate via network.
- Easy to scale increase capacity.

Shared disk: Each DB has own CPU, mem,
- share disk.
- Messages between CPU's to learn about current state

Shared Memory: Each DB has own CPU
- Share Mem, disk
- parallel computing

shared memory → hard
requires hardware & software support.

Sharding: data partitioning

hash partitioning:

choose partitioning attributes

hash function h w/ range $0, \dots, n-1$

range partitioning:

choose attribute

partition vector $[v_0, \dots v_{n-2}]$

↑ partition value of tuple

nodes = n+1

n = num of vectors

$v_i \leq v_{i+1} \rightarrow$ node $i+1$

$v < v_0 \rightarrow$ node 0 (note)

$v \geq v_{n-2} \rightarrow$ node $n-1$

Ex: vector [5, 11]

at(2) → $2 < 5$ → go to 0

at(8) → $8 \leq 9$ → go to 2

at(20) → $20 \geq 11$ → node 2

$$\begin{bmatrix} 15 \\ 40 \\ 75 \end{bmatrix}$$

→ node 1 $(-\infty, 15)$

→ node 2 $[15, 40)$

→ node 3 $[40, 75)$

→ node 4 $[75, +\infty)$

$41 \leq 42$

Hash part: point queries on part attri
can lookup single nodes, others for answer queries

range queries: must be processed at all nodes.

Range part: good for sequential scan

point queries: only one node accessed

range queries: remaining nodes are available for other queries.

Distributed Query Processing

db: collection of interrelated items.

dbms: software manages, stores, queries, analyze dbs

SQL: simple query language.

specify "what" but not "how"
db how to answer query in most efficient way] declarative language.

Java/C++ procedure language: clearly specify steps

db → stores tables

tables → set of attributes (columns)

attribute has type

domain: set of allowed values for each of attribute attribute

null value of any type.

data is stored as collection of tuples (rows)
(order does not matter)

attribute must be atomic (single values)

schema of table: set of attris. and types

instance: actual contents in the table at a given time.

key: unique value in each tuple (row) or set of attris
whose combined values are unique.

Candidate key: min. subset of attris that uniquely identify a tuple in the table.

primary key: chosen key

super key: super set of candidate key

DBMS automatically generates unique keys.

Foreign key: set of attris. in relation A, used to refer to a tuple in B.

Data integrity: constraints on dbs to prevent errors

entity: primary key cannot be null

referential: ..   user: max gpa is 4.0, name can't be null

Data definition language (DDL)
Data Manipulation language (DML)

CREATE DATABASE "name"
USE "name";
CREATE TABLE   DROP TABLE   ALTER TABLE
(add/delete column/constraint)

Constraints:

CREATE TABLE "name"
column 1 datatype [constraint];

common:
NOT NULL, UNIQUE,
PRIMARY KEY, Foreign Key

[table constraint] also can

```
sid     CHAR (10)  PRIMARY KEY
sname   CHAR(30)   NOT NULL
age     INT
PRIMARY KEY (sid)
```

add new col:
ALTER TABLE Student ADD
address CHAR (100);
↓
DROP: remove

ALTER COLUMN

INSERT, DELETE, UPDATE, SELECT

- INSERT: data into table
- DELETE: all records of DB
- UPDATE: Existing data within a table
- SELECT: retrieve data

if empty ⇒ null

INSERT INTO Student VALUES ('s101', ...);

QUERIES:

SELECT A1, ..., An ← attributes
FROM T1, ..., Tm ← tables
WHERE P ← predicates or condition

[EXPRESSION] ⤳ output table

SELECT sid, sname
FROM student;

SELECT * ← for all
FROM student;

WHERE cid='cs290' OR, AND, NOT

DISTINCT Remove duplicates

LIMIT n, show first n results

WHERE cid LIKE 'CS%'; %: ← match string of any length

NOT LIKE

IS NULL

NOT NULL

_: match a single char

ORDER BY attributes DESC; ASC

col 1 ASC, col 2 DESC;

Find total: SELECT COUNT(*), COUNT(DISTINCT(sid))

FROM STUDENT

AVG(grade), MAX(), MIN(), SUM()

Group by: groups rows that have the same values into summary rows

# of students enrolled for each course:

CS 240: 40

CS 242: 20

⋮

Q_1
SELECT student, count(student)
FROM table
GROUB BY course;

same but now only show courses with at least 3 students

Q_2
Q_1
HAVING COUNT(student) >3;

HAVING for group by

WHERE for table

AGG fcns can't be used in WHERE clause.

Q_2

ORDER BY AVG(grade) ASC;

For multiple tables:

```
SELECT attributes ← of multiple tables
FROM table, table2, ..
WHERE table.id = table2.id AND table2.course ='MA453'
 AND table2.grade > 3.5;
```

NESTED

INNER QUERY → CHILD

OUTER QUERY → Parent QUERY

Students in the same dept as Susan.

```
SELECT sid, sname, dept
FROM Student
WHERE dept IN ← also =
      (SELECT dept FROM Student WHERE 'sname' = 'susan');
```

single values: "=", ">", "<"

multiple

> ANY: bigger than some value

> ALL: bigger than all value

:

```
WHERE < ALL
      (Query)
```

SET operations:

```
Q1
UNION / UNION ALL / Intersect / Except
Q2;
```

↑ (UNION ALL) keep duplicates

Row update:

```
UPDATE Student
SET dept = 'CS'
WHERE sid = 'S102';
```

```
UPDATE STUDENT
SET age = age+1;
```

```
INSERT
INTO table (col1,...)
 Query;
```

insert this ↓

```
INSERT
INTO
Dept_Age (dept, avg_age)
SELECT dept
FROM Student
Group by dept;
```

DELETE del Rows
FROM Table
WHERE conditions;

data file stored on disk on pages ("blocks")

usually page size is 4KB

I/O : read page disk → memory
write page

random I/O : read a page at random
seek + read
dom

B-tree
index traversal

DBMS stores table as file on disk
↓
collection of pages

column store : store table col by col
Decomposition storage model (DSM)

col store for wide table (10s to 100s)

I/O cost : # of reads/write for task

I/O time : time to perform I/O

fread() : read time
fwrite(), fseek()

sequential I/O : read large chunk of data

faster, read only, scan table

(NSM) n-ary storage model
row store : store rows 1 by 1

Heap file : records can be stored anywhere with free space.

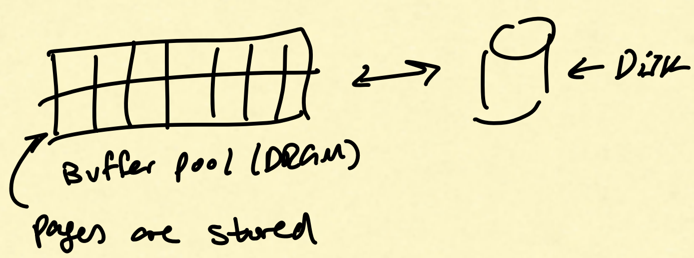
keep track of free space w/
linked list / free space map
↓
array w/ 1 entry per page

col store → OLAP (analytical processing)
Row store → OLTP (transactional processing)
} database workloads

OLTP:
simple queries read/update small amount of data, entity
mostly update/write
book flight ticket

OLAP:
complex queries
mostly read only
analyze what customers/regions bought

Buffer Management



index: speedy retrieval of data from an underlying table.

B+ tree index
point Q, range Q

Hash index
can't range Q
poin Q efficient

Hash table
array of size m entries

$H(x) = x \% m$

collision: $H(k_1) = H(k_2)$

space $O(n)$

Q cost: Avg $O(1)$

Hash table in DB

Buckets on disk (pages)

Directory in memory

Avg query cost $O(n/m)$, $O(1)$, $m \approx n$
(n: keys, m: buckets)

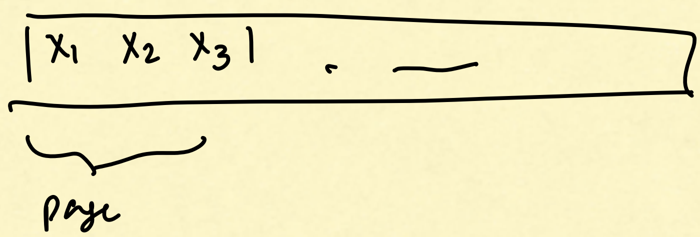
B tree
search, insert, del
$O(\log_2(n))$

mem-based index:
best: data in memory
Sub-optimal: data stored in disk.

B+ tree disk optimized tree w/ $O(\log_B N)$ I/O ← Insert, delete

B is page fan out

node ∈ disk block



leaf node has $\lceil B/2 \rceil$ to B elements where B is at least 3, 2 or 3

each internal node has $\lceil B/2 \rceil$ to B child nodes

leaf: store data, internal: only for routing

node values: ascending order, left subtree values < right subtrees

Point Q I/O cost: $O(\log_B N)$ (N: total num of elts)

cost of range Q
M: # tuples for range query
R: # of tuples per page
$O(\lceil \frac{M}{R} \rceil)$ clustered index
$O(M)$ unclustered

Overflows → more values than expected

1) split into halves v_1, v_2

insert value to parent of v

if overflows split parent

add to parent ...