Exercise Sheet: Data Storage

Data, Data Storage, Data Collection

Exercise 1 – Hashing a New Fleet

The operator keeps the state of each bike, `bike_id` → {`station`, `battery`, `status`}, in a hash table of m slots, with chaining: each slot holds the list of the entries that land there. The key of a bike is its number (bike `B12` has key $k = 12$). As in Python, the hash of an integer is the integer itself, $\text{hash}(k) = k$, so the slot of a key k is

$$\text{hash}(k) \bmod m = k \bmod m.$$

A lookup computes the slot of the key, then walks the list of that slot and compares the key of each entry until it finds the bike.

(**Q1**) A new fleet arrives, numbered by tens: `B10`, `B20`, ..., `B100`. Place the ten keys in a table with $m = 10$, then in a table with $m = 7$. In each table, how many entries may a lookup of `B50` have to compare?

With $m = 10$, every key is a multiple of 10, so every key lands in slot 0. With $m = 7$:

k	10	20	30	40	50	60	70	80	90	100
$k \bmod 7$	3	6	2	5	1	4	0	3	6	2

Slots 0, 1, 4 and 5 hold one key, slots 2, 3 and 6 hold two.
With $m = 10$, the lookup of `B50` walks the chain of slot 0, which holds all ten bikes: up to 10 comparisons, the $\mathcal{O}(n)$ worst case of a hash table. With $m = 7$, `B50` is alone in slot 1: one comparison. The keys are the same and the table with $m = 10$ is even larger: only the choice of m changed.

(**Q2**) The keys of the new fleet are $k_j = 10j$, for $j = 1, 2, \ldots$ Let d be the greatest common divisor of 10 and m. Show that $k_j \bmod m$ is always a multiple of d, and deduce the largest number of slots that the keys can reach.

Write $k_j \bmod m = 10j - qm$, where q is the quotient of $10j$ by m. Both $10j$ and qm are multiples of d, so their difference is too. The slot is a multiple of d between 0 and $m-1$, and there are m/d of them: the keys reach at most m/d slots (and one can show that they reach all of them as j grows).

(**Q3**) How many slots can the keys reach for $m = 1$, 7, 8, 10, 11 and 12? Which m would you choose?

- $m = 1$: $d = 1$, a single slot, whatever the keys.
- $m = 7$: $d = 1$, all 7 slots.
- $m = 8$: $d = 2$, 4 slots out of 8 $(0, 2, 4, 6)$.
- $m = 10$: $d = 10$, a single slot (0), as in the previous question.
- $m = 11$: $d = 1$, all 11 slots.
- $m = 12$: $d = 2$, 6 slots out of 12.

Choose $m = 11$: the keys reach every slot, and the most slots. The case $m = 1$ shows that $d = 1$ is not enough: what counts is m/d, so the table must also be large enough

for its keys.

(**Q4**) Real keys often follow a step: numbers by tens, even identifiers, sizes in powers of two. Why are table sizes often chosen prime?

With a step s, the same argument shows that the keys reach at most $m/\gcd(s, m)$ slots: the table spreads them only if m shares no factor with s. A prime m shares a factor with s only if s is a multiple of m, so it suits every step except its own multiples, whereas $m = 8$ or $m = 12$ fails with every even step.
As a sidenote, tables whose size is a power of two (Java's `HashMap`, CPython's `dict`) instead mix the high bits of the hash into the slot computation, so that keys sharing their low bits still spread.

From now on, we assume *uniform hashing*: each key lands in each of the m slots with probability $1/m$, independently of the other keys.

(**Q5**) Index the keys $1, \dots, n$. For each pair $i < j$, let $X_{ij} = 1$ if keys i and j land in the same slot and $X_{ij} = 0$ otherwise, and let $N = \sum_{i<j} X_{ij}$ be the number of colliding pairs. Show that

$$\mathbb{E}[N] = \binom{n}{2}\frac{1}{m},$$

and compute it for $n = 20$ and $m = 1000$. Are the X_{ij} independent? Does it matter?

Keys i and j both land in slot s with probability $1/m^2$, and there are m slots:

$$\mathbb{E}[X_{ij}] = \Pr(X_{ij} = 1) = m \times \frac{1}{m^2} = \frac{1}{m}.$$

By linearity of expectation, summing over the $\binom{n}{2}$ pairs,

$$\mathbb{E}[N] = \sum_{i<j} \mathbb{E}[X_{ij}] = \binom{n}{2}\frac{1}{m} = \frac{n(n-1)}{2m}.$$

For $n = 20$ and $m = 1000$: $\mathbb{E}[N] = 190/1000 = 0.19$.
The X_{ij} are not independent: if $X_{12} = 1$ and $X_{13} = 1$, then $X_{23} = 1$. It does not matter: linearity of expectation holds for any random variables, independent or not.

(**Q6**) Insert the keys one by one. Show that the probability that no two keys collide is

$$\prod_{i=0}^{n-1}\left(1 - \frac{i}{m}\right),$$

and compute the probability of at least one collision for $n = 3$ and $m = 10$.

If the first i keys did not collide, they occupy i distinct slots. Key $i + 1$ avoids them with probability $(m - i)/m = 1 - i/m$, whatever happened before. Multiplying these conditional probabilities for $i = 0, \dots, n - 1$ gives the product.
For $n = 3$ and $m = 10$:

$$\Pr(\text{no collision}) = 1 \times \frac{9}{10} \times \frac{8}{10} = 0.72, \qquad \Pr(\text{at least one collision}) = 1 - 0.72 = 0.28.$$

We consider that a table keeps its load factor $\alpha = n/m$ bounded by moving to a larger array when α gets too high.

(**Q7**) Write $\mathbb{E}[N]$ in terms of n and α. How many colliding pairs do you expect for $n = 1\,000\,000$ keys and $\alpha = 1/2$?

$\mathbb{E}[N] = \frac{n(n-1)}{2m} = \frac{(n-1)\alpha}{2} \approx \frac{n\alpha}{2}$. For $n = 10^6$ and $\alpha = 1/2$: about 250 000 colliding pairs.

(**Q8**) Show that the expected number of other keys in the slot of a given key is $(n-1)/m$, which is less than α.

The number of other keys in the slot of key i is $\sum_{j \neq i} X_{ij}$. By (**Q5**), each term has expectation $1/m$, so the sum has expectation $(n-1)/m < n/m = \alpha$.

(**Q9**) Do your findings contradict a lookup in $\mathcal{O}(1)$ on average? What should trigger a resize: the number of collisions or the load factor?

No. A lookup walks the chain of a single slot: on average at most $1 + \alpha$ entries, a constant as long as α stays bounded, whatever n. The total number of collisions grows with n, but each lookup only pays for its own slot. So the resize is triggered by the load factor: collisions cannot be avoided, only kept short.

Exercise 2 – Two Layouts for the Trips

The operator stores its trips in a table of 1 000 trips, `T1` to `T1000`, with 10 columns:

`trip_id`, `rider_id`, `bike_id`, `start_station`, `end_station`, `start_time`, `end_time`, `start_hour`, `minutes`, `price`.

We assume that every value takes the same space, and that the disk reads and writes *blocks* of 100 values. Row-wise, a block holds 10 complete trips; column-wise, a block holds 100 values of a single column. The trips are stored in the order of `trip_id`, so the position of a trip is known without searching.

(**Q1**) For each layout, how many values must be read to answer the query "average `minutes` per `start_hour`"? And the query "show trip `T512`"? Which of these values are contiguous?

- Average duration per start hour. Row-wise: every trip must be read, and each trip comes with its 10 values, so 10 000 values, of which 8 000 are useless. Column-wise: only the two columns, so 2 000 values, in two contiguous runs.
- Trip `T512`. Row-wise: its 10 values, contiguous. Column-wise: the same 10 values, but at position 512 of each of the 10 columns, so in 10 different places.

(**Q2**) Count the blocks read by each query in each layout. More generally, a query reads c of the C columns of all n records, in blocks of B values: how many blocks does it read in each layout? What decides the ratio?

- Average duration per start hour. Row-wise: $1\,000/10 = 100$ blocks. Column-wise: each column fills $1\,000/100 = 10$ blocks, so 20 blocks. The column layout reads 5 times fewer blocks.
- Trip `T512`. Row-wise: 1 block (the one holding `T511` to `T520`). Column-wise: 10 blocks, one per column. The row layout reads 10 times fewer blocks.

In general, a row-wise block holds B/C whole records, and a column-wise block holds B values of one column.

- All n records: row-wise, every block is read, so nC/B blocks whatever c; column-wise, only the c columns, so cn/B blocks. Columns win by C/c.
- A single record: 1 block row-wise (if it fits in a block), and c blocks column-wise, one per column read. Rows win by c, that is C for a whole record.

The ratio is decided by the share of the columns read, and by whether the query needs all the records or only a few.

Suppose we change the workload and the unlock app inserts a new trip at every unlock, and at every return updates `end_station`, `end_time`, `minutes` and `price` of that trip.

(**Q3**) How many blocks does one insert write in each layout? One return?

An insert writes 1 block row-wise (the last block, where the new trip is appended) and 10 blocks column-wise (the end of each column). A return writes 1 block row-wise (the block of the trip) and 4 blocks column-wise (one per updated column).

(**Q4**) The app makes thousands of unlocks and returns per hour; the city analysts compute the average duration per start hour once a day, over the whole history. Which layout suits each of them? How can the operator serve both?

The app makes many small writes and reads by key: rows (OLTP). The analysts scan two columns of the whole history: columns (OLAP). The operator keeps the operational database in rows and periodically copies the trips into a columnar store, for example Parquet files (extract, transform, load). Each copy is laid out for its own workload, at the cost of storing the trips twice and of analyses that lag behind the latest trips.

Exercise 3 – A Night of Rebalancing

At night, a truck moves 5 bikes from station `007` (20 bikes) to station `012` (10 bikes). The database records the move as one transaction T_1, while the city dashboard runs a transaction T_2 that displays the total number of bikes docked at the two stations. We write $R_1(\texttt{007})$ when T_1 reads the record of station `007`, $W_1(\texttt{007})$ when it writes it, and C_1 for its `COMMIT`:

$$T_1: \quad R_1(\texttt{007}) \quad W_1(\texttt{007}) \leftarrow 15 \quad R_1(\texttt{012}) \quad W_1(\texttt{012}) \leftarrow 15 \quad C_1$$
$$T_2: \quad R_2(\texttt{007}) \quad R_2(\texttt{012}) \quad C_2 \quad \text{(displays the sum of the two values read)}$$

The DBMS may interleave the operations of the two transactions. Here are three interleavings, called *schedules*, with time running downwards:

S1

	T_1	T_2
1	$R_1(\texttt{007})$	
2	$W_1(\texttt{007})$	
3		$R_2(\texttt{007})$
4		$R_2(\texttt{012})$
5		C_2
6	$R_1(\texttt{012})$	
7	$W_1(\texttt{012})$	
8	C_1	

S2

	T_1	T_2
1	$R_1(\texttt{007})$	
2	$W_1(\texttt{007})$	
3		$R_2(\texttt{007})$
4	$R_1(\texttt{012})$	
5	$W_1(\texttt{012})$	
6	C_1	
7		$R_2(\texttt{012})$
8		C_2

S3

	T_1	T_2
1		$R_2(\texttt{007})$
2	$R_1(\texttt{007})$	
3	$W_1(\texttt{007})$	
4	$R_1(\texttt{012})$	
5	$W_1(\texttt{012})$	
6	C_1	
7		$R_2(\texttt{012})$
8		C_2

(**Q1**) Run the two *serial* schedules, T_1 then T_2, and T_2 then T_1. What does the dashboard display? Recall that isolation means that concurrent transactions run as if one after the other.

30 in both cases: $15 + 15$ if T_1 runs first, $20 + 10$ if T_2 runs first. Isolation does not say which transaction comes first, only that the result must be that of *some* serial order: here, any schedule must make the dashboard display 30.

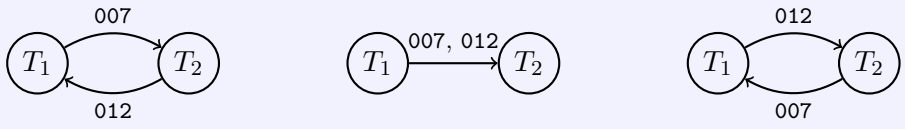
(**Q2**) What does the dashboard display under S1, S2 and S3? Which schedules respect isolation? In which schedules does T_2 read only committed values? Is that enough to ensure isolation?

- S1: $15 + 10 = 25$. T_2 reads station `007` after T_1's write and station `012` before it: five bikes vanish.
- S2: $15 + 15 = 30$, the result of T_1 then T_2, although the two transactions are interleaved.
- S3: $20 + 15 = 35$. T_2 reads station `007` before T_1's write and station `012` after it: five bikes are counted twice.

Only S2 respects isolation. T_2 reads only committed values in S3: in S1 and S2, it reads station `007` after $W_1(\texttt{007})$ and before C_1. So reading committed values is not enough: in S3, each value was committed, but the two values come from two different states of the database. Nor is it necessary, since S2 reads an uncommitted value and is correct.

Two operations *conflict* if they belong to different transactions, access the same record, and at least one of them is a write. The *precedence graph* of a schedule has one node per transaction, and an edge $T_i \to T_j$ whenever an operation of T_i conflicts with a later operation of T_j. One can show that a schedule can be turned into a serial one, by swapping neighbouring operations that do not conflict, exactly when its precedence graph has no cycle; the serial order is then given by a topological order of the graph.

(**Q3**) Draw the precedence graphs of S1, S2 and S3, and check them against your answer to (**Q2**). For each schedule whose graph has no cycle, give an equivalent serial order.



S1 S2 S3

- S1: $W_1(\texttt{007})$ comes before $R_2(\texttt{007})$, so $T_1 \to T_2$; $R_2(\texttt{012})$ comes before $W_1(\texttt{012})$, so $T_2 \to T_1$. Cycle.
- S2: both conflicts give $T_1 \to T_2$ ($R_1(\texttt{012})$ and $R_2(\texttt{012})$ do not conflict). No cycle: the serial order is T_1 then T_2.
- S3: $R_2(\texttt{007})$ before $W_1(\texttt{007})$ gives $T_2 \to T_1$; $W_1(\texttt{012})$ before $R_2(\texttt{012})$ gives $T_1 \to T_2$. Cycle.

This matches (**Q2**): the two schedules with a wrong total have a cycle, the correct one does not. Essentially, we turned "as if one after the other" into a test on a graph.

(**Q4**) Explain why swapping two conflicting operations may change the result, why two reads never conflict, and why a cycle in the precedence graph rules out every serial order.

Swapping a read and a write of the same record changes the value read; swapping two writes of the same record changes the value left in the database. Two reads return the same values in either order, so their order does not matter.
Swapping neighbouring operations that do not conflict never changes the order of two conflicting ones, so every edge survives the swaps. In a serial schedule, all the operations of a transaction come before those of the next one, so every edge goes from an earlier transaction to a later one.
A cycle $T_1 \to T_2 \to T_1$ would require T_1 to come before T_2 and T_2 before T_1: no serial order fits.

A classic way for a DBMS to enforce isolation is *strict two-phase locking*, which extends the rule of the lecture to reads. A transaction locks a record before accessing it:

- a *shared* lock, to read the record: several transactions may hold one on the same record at once, since two reads do not conflict;
- an *exclusive* lock, to write the record: a transaction gets it only if no other transaction holds a lock of either kind on the record. A transaction that holds a shared lock may turn it into an exclusive one under the same condition.

A transaction that cannot get a lock waits until it is released. Locks are released only at `COMMIT`, all at once. The name says that a transaction first only acquires locks, then only releases them (two phases), and that the release happens at `COMMIT` (strict).

(**Q5**) Replay the schedules that do not respect isolation with these locks: who waits, and when? What does the dashboard display, and which serial order do you get?

- S1: T_1 takes a shared lock on station `007` to read it, then turns it into an exclusive lock to write it. At step 3, T_2 cannot get its shared lock on `007` and waits. T_1 reads and writes `012` and commits, releasing its locks. T_2 then reads 15 and 15: the dashboard displays 30, the result of T_1 then T_2.
- S3: T_2 holds a shared lock on `007`, and T_1 gets one too. At step 3, T_1 needs an exclusive lock on `007`, which it cannot get while T_2 holds a shared one: T_1 waits. T_2 reads 10 for station `012` and commits: the dashboard displays 30. T_1 then resumes. The result is that of T_2 then T_1.

Two reads of the same record proceed together (both transactions read `007` in S3). A conflicting access, however, waits until the transaction that holds the lock commits. So every edge $T_i \to T_j$ of the precedence graph means that T_i committed before T_j: the edges follow the commit order and cannot form a cycle.
In S2, T_2 reads `007` before T_1 commits; the result is correct only because T_1 does commit. With locks, T_2 would wait at step 3: locking accepts fewer schedules than the graph test, because it also protects against an abort.