
Question 1 [30 pts]

You've been appointed as the database designer of a smart-farming application. The application is supported by four types of on-field sensors:

(A) location sensors denoting the position of different soil locations in the field,

(B) dielectric soil moisture sensors that report the moisture content (in %) and hence the level of precipitation of a location,

(C) electrochemical sensors for soil nutrient level (in % of organic matter) of a location, and

(D) optical sensors that report condition of the clay, natural matter, and humidity properties of a soil location

As a database designer, you have a file with raw data collected from the field. The file contains 500M entries each consisting of the values of *A, B, C,* and *D* (i.e., the data attributes defined in the previous paragraph). Your task is to create a database *soil_db* with this data. Your job is to analyze the data to decide for which soil locations an urgent inspection or a standard maintenance is required. The rationale for the analysis is as follows: if the soil moisture is between 20-60% and soil nutrient is less or equal to 5%, it is considered standard. Anything that is not standard, needs an urgent inspection.

$\textbf{Q1.}$ **What would you query *soil_db* to determine the soil locations where an urgent inspection is needed? Write the SQL query. [10 pts]**

```
SELECT A FROM soil_db WHERE (B < 20 OR B > 60) AND C > 5
```

$\textbf{Q2.}$ **How would you design the storage layout of *soil_db*? Justify this choice of data layout with (i) a brief text explaining your assumptions (if any) and intuition and (ii) a graphical illustration using boxes or other representative shapes to describe the layout (rows, columns, pages, etc). [10 pts]**

There are two assumptions which would impact our design decision:

1. The sensors in the farm continuously log data (not just one time for us to analyze later)
2. Historical data should be persisted (perhaps for longer time scale data analytics)

If the second condition is not true, we can keep our single table row/column store:

```
        tbl_sensor_data

id    _____  _____  _____  _____
     |  A  ||  B  ||  C  ||  D  |
     |_____||_____||_____||_____|
  0  |xxx| |xxx| |xxx| |xxx|
  1  |xxx| |xxx| |xxx| |xxx|
  2  |xxx| |xxx| |xxx| |xxx|
  3  |xxx| |xxx| |xxx| |xxx|
    4   │xxx│  │xxx│  │xxx│  │xxx│
    5   │xxx│  │xxx│  │xxx│  │xxx│
        │   │  │   │  │   │  │   │
        │   │  │   │  │   │  │   │
        └───┘  └───┘  └───┘  └───┘
```

Then if the first condition is also true, it would be quite easy to update data points when we receive new data; we just update the single entry in *B, C,* and *D* corresponding to the relevant location in *A*.

If the second condition were true, meaning we would also need to persist historical data, things become a bit more complex. Now a single sensor location corresponds to multiple data points, so we could use two tables, like so:

```
 tbl_sensor_locations                 tbl_sensor_data

    ┌───┐  ┌────┐          ┌────────┐  ┌─────┐  ┌───┐  ┌───┐  ┌───┐
    │ A │  │ id │          │ loc_id │  │ time│  │ B │  │ C │  │ D │
    ├───┤  ├────┤          ├────────┤  ├─────┤  ├───┤  ├───┤  ├───┤
    │xxx│  │ 00 │          │   00   │  │ 1:00│  │xxx│  │xxx│  │xxx│
    │xxx│  │ 01 │          │   01   │  │ 1:00│  │xxx│  │xxx│  │xxx│
    │xxx│  │ 02 │          │   02   │  │ 1:00│  │xxx│  │xxx│  │xxx│
    │xxx│  │ 03 │          │   00   │  │ 2:00│  │xxx│  │xxx│  │xxx│
    │xxx│  │ 04 │          │   03   │  │ 2:00│  │xxx│  │xxx│  │xxx│
    │xxx│  │ 05 │          │   04   │  │ 2:00│  │xxx│  │xxx│  │xxx│
    │   │  │    │          │   05   │  │ 2:00│  │xxx│  │xxx│  │xxx│
    │   │  │    │          │        │  │     │  │   │  │   │  │   │
    └───┘  └────┘          │        │  │     │  │   │  │   │  │   │
                           │        │  │     │  │   │  │   │  │   │
                           │        │  │     │  │   │  │   │  │   │
                           │        │  │     │  │   │  │   │  │   │
                           └────────┘  └─────┘  └───┘  └───┘  └───┘
```

This might not be optimal, and depending on how much historical data we must store and the size of the *A* data type, we might replace the *loc_id* in *tbl_sensor_data* with *A* itself. This has the penalty of having many duplicates in this column, but this might not be so bad depending on other considerations.

$\textbf{Q3.}$ **Would your answer in Q2 change if the application requirement is to run this query (Q1) several times a day and you are to optimize for performance? Please explain the new layout in detail. We do not expect a detailed cost analysis here but rather an intuitive explanation. [10 pts]**

For this answer, we assume that second condition from the previous part does not hold, since optimizing the table would require some concepts which we have not covered yet. So suppose we had a single table *tbl_sensor_data* as in the previous part. We can assume the first condition is true, since otherwise the table data wouldn't change and so we could just perform the query once and use the answer for the rest of the queries.

One of the main bottlenecks in the query is that we need to perform two range select operators on *B* and *C* respectively, which might incur a decently high data movement cost. To lower this cost, we could pre-compute the result of this selection into a new bit-vector column *E*, which stores whether the sensor requires immediate attention. So our new table looks like:

```
        tbl_sensor_data
        ┌───┐ ┌───┐ ┌───┐ ┌───┐ ┌───┐
 id     │ A │ │ B │ │ C │ │ D │ │ E │
        ├───┤ ├───┤ ├───┤ ├───┤ ├───┤
  0     │xxx│ │xxx│ │xxx│ │xxx│ │ 0 │
  1     │xxx│ │xxx│ │xxx│ │xxx│ │ 1 │
  2     │xxx│ │xxx│ │xxx│ │xxx│ │ 0 │
  3     │xxx│ │xxx│ │xxx│ │xxx│ │ 1 │
  4     │xxx│ │xxx│ │xxx│ │xxx│ │ 1 │
  5     │xxx│ │xxx│ │xxx│ │xxx│ │ 0 │
        │   │ │   │ │   │ │   │ │   │
        │   │ │   │ │   │ │   │ │   │
        └───┘ └───┘ └───┘ └───┘ └───┘
```

Now every time we update a sensor's data, we also set *E* with the value `(B < 20 OR B > 60) AND C > 5`. Then the query from Q1 becomes:

```
SELECT A FROM tbl_sensor_data WHERE E = 1
```

Since *E* is a bit-vector, this decreases data movement during selection by a factor of:

```
(sizeof(B)+sizeof(C)) / (sizeof(E) / 8) = 8 * (sizeof(B) + sizeof(C))
```

So if *B* and *C* are both 4-byte ints, we would have to move 64 times less data during selection! This greatly increases the speed of the query. Of course, this moves the cost of computing the query to the insertion operation, since we have to update *E* every insert anyways, but this isn't that bad since *B* and *C* would already be loaded up during insert anyways, so the effect is negligible.

Another option if we don't want to add another column would be to merge *B* and *C* into a single column:

```
        tbl_sensor_data
        ┌───┐ ┌───────┐ ┌───┐
 id     │ A │ │ B - C │ │ D │
        ├───┤ ├───────┤ ├───┤
  0     │xxx│ │xxx xxx│ │xxx│
  1     │xxx│ │xxx xxx│ │xxx│
  2     │xxx│ │xxx xxx│ │xxx│
  3     │xxx│ │xxx xxx│ │xxx│
  4     │xxx│ │xxx xxx│ │xxx│
  5     │xxx│ │xxx xxx│ │xxx│
        │   │ │       │ │   │
        │   │ │       │ │   │
        └───┘ └───────┘ └───┘
```

Since we always access *B* and *C* together, this might decrease data movement for single entry updates, since we now only load a single page of the *B*-*C* column instead of two pages for *B* and *C* respectively. This also would make the query faster since any acceleration structure on *B*-*C* could sort/index by both *B* and *C* which would allow it to filter with less data movement since it can reject entire *B*-*C* blocks without having to check *B* and *C* independently.

Question 2 [30 pts]

There is a database table T with eight columns $C_1, C_2,\cdots ,C_8$. All columns contain unsigned 32 bit integers uniformly distributed in $[0, 2^{32}-1]$. There are 2^{30} rows in the table. The table data resides on disk.

$\textbf{Q1.}$ **[6X3 pts] Assume the following SQL queries:**

```
S1: SELECT AVG(C5) FROM T WHERE C4 > 90
S2: SELECT C1,C5 FROM T WHERE C2 = 120 AND C3 > 339
S3: SELECT * FROM T WHERE C2 + C5 > 3
```

Write :

- **the query plans, and**
- **compute the minimal number of memory misses. Do this for a pure columnar layout and pure row-oriented layout.**

Define the operators you use in the plans and briefly describe how they work. You do not need to write code or pseudo-code for the operators.

For this problem, our operators will **not** be vectorized for the sake of simplicity.

Let's begin with the first query S_1:

```
// S1: SELECT AVG(C5) FROM T WHERE C4 > 90
s = select(T.C4, greater_than(90))
f = fetch(T.C5, s)
a = average(f)
output(a)
```

Here the operator definitions are fairly straightforward:

- `select` takes in a column and some filter object (e.g. `greater_than(90)` means `select entries strictly greater than 90) and outputs an array of indices corresponding to the selection results.
- `fetch` takes in a column and a selection result and outputs the array of entries in the column which have the selection indices.

- `average` takes in a fetch output or a column and returns a floating point representing the average of the data in the list.
- `output` outputs any time of object in the query plan to the client. (It may print, save somewhere etc...)

Now let's compute the memory misses when we run this query. We'll make the following assumptions about our system:

- No other processes are running on the core. (so main memory and the cache are all ours)
- Main memory and the cache start out empty.
- The code + runtime of the database server take up a negligible amount of main memory and cache space.

Lastly, we note that for the row layout, the select operator would load all 8 columns in at once, causing $8*2048=8192$ memory misses. Since this is 32 Gb of data, it fits into memory so further operators do not result in any more memory misses. This means that for each of these queries, each query results in 8192 memory misses.

Columnar Layout for S_1:

Let's start with the select operator. First observe that the size of each column is $4\times 2^{30}=2^{32}$ or 4 gigabytes. This corresponds to 2048 disk pages.

Next, notice that the selection asks for integers greater than 90. Since the data is uniformly distributed in $[0,2^{32}-1]$, this means we have a selectivity of around $100%$ so our selection result will consist of $4\times 2^{30}=2^{32}$ bytes of data assuming 4 byte virtual ids. So we reserve 4 Gb for this selection data, which doesn't affect memory misses.

To fetch, we similarly must load 2048 disk pages corresponding to C_5, since the selectivity was practically $100%$. Thus we have 4096 memory misses.

Next, let's do this for S_2.

```
// S2: SELECT C1,C5 FROM T WHERE C2 = 120 AND C3 > 339
s1 = select(T.C2, equals(120))
s2 = select(T.C3, s2, greater_than(339))

f1 = fetch(T.C1, s2)
f5 = fetch(T.C5, s2)

output(f1, f2)
```

Columnar Layout for S_2:

The first selection misses 2048 pages but selects a very small set of values, indeed the expected size of the selection is 0.25. Then the second selection must fetch 0.25 disk pages. (Here we're working in the average case) The condition >339 has a very high selectivity so the expected size of s_2 is still 0.25. Finally, we load 0.50 disk pages in the fetch operations and we are done. So we have about ≈ 2048 disk misses loaded on average.

Finally, let's do this for S_3.

```
// S3: SELECT * FROM T WHERE C2 + C5 > 3
f = sum(T.C2,T.C5)
s = select(f, greater_than(3))

f1 = fetch(T.C1, s)
f2 = fetch(T.C2, s)
f3 = fetch(T.C3, s)
f4 = fetch(T.C4, s)
f5 = fetch(T.C5, s)
f6 = fetch(T.C6, s)
f7 = fetch(T.C7, s)
f8 = fetch(T.C8, s)
output(f1, f2, f3, f4, f5, f6, f7, f8)
```

Columnar Layout for S_3:

In this case, note that the selectivity is basically $100%$, so all the columns must be fully loaded. Since they all fit into main memory, we have 8192 memory misses.

Summary:

Overall, we have the following table of memory misses

Query	Columnar Layout Misses	Row Layout Misses
S_1	4096	8192
S_2	2048	8192
S_3	8192	8192

$\textbf{Q2.}$ **[4X3 pts] For each query in $\textbf{Q1}$:**

- **design a hybrid data layout using column groups.**
- **compute the number of memory misses in the hybrid layout.**
- **discuss the benefit of the column group layout in terms of CPU cache utilization.**

```
S1: SELECT AVG(C5) FROM T WHERE C4 > 90
S2: SELECT C1,C5 FROM T WHERE C2 = 120 AND C3 > 339
S3: SELECT * FROM T WHERE C2 + C5 > 3
```

Since we're optimizing for CPU cache utilization, we want tight loops which use all of the data they fetch from main memory. For S_1, there isn't much we can do without changing our operators.

```
┌──┐ ┌──┐ ┌──┐ ┌──┐ ┌──┐ ┌──┐ ┌──┐ ┌──┐
│C1│ │C2│ │C3│ │C4│ │C5│ │C6│ │C7│ │C8│
```

```
├──┤ ├──┤ ├──┤ ├──┤ ├──┤ ├──┤ ├──┤ ├──┤
│  │ │  │ │  │ │xx│ │xx│ │  │ │  │ │  │
│  │ │  │ │  │ │xx│ │xx│ │  │ │  │ │  │
│  │ │  │ │  │ │xx│ │xx│ │  │ │  │ │  │
└──┘ └──┘ └──┘ └──┘ └──┘ └──┘ └──┘ └──┘
```

If we change up some of our operators, we could merge C_4 and C_5 to get:

```
┌──┐ ┌──┐ ┌──┐ ┌─────┐ ┌──┐ ┌──┐ ┌──┐
│C1│ │C2│ │C3│ │C4-C5│ │C6│ │C7│ │C8│
├──┤ ├──┤ ├──┤ ├─────┤ ├──┤ ├──┤ ├──┤
│  │ │  │ │  │ │xxxxx│ │  │ │  │ │  │
│  │ │  │ │  │ │xxxxx│ │  │ │  │ │  │
│  │ │  │ │  │ │xxxxx│ │  │ │  │ │  │
└──┘ └──┘ └──┘ └─────┘ └──┘ └──┘ └──┘
```

But in order to efficiently utilize this we'd need an operator such as:

```
float average = 0.0;
for (size_t i = 0; i < num_rows; ++i) {
    if (C4[i] > 90) {
        average += (float) C5[i] / (float) num_rows;
    }
}
return average;
```

Such a merge wouldn't affect the number of memory misses since the selectivity is high and we have to bring all of the data anyways.

It would be better from a cache perspective though because it eliminates the need for a selection array which will reduce cache misses. Since the selectivity is extremely high, this is non negligible.

For S_2, a natural thing to do would be to merge C_1, C_5 and C_2, C_3 to get:

```
┌─────┐ ┌─────┐ ┌──┐ ┌──┐ ┌──┐ ┌──┐
│C1-C5│ │C2-C3│ │C4│ │C6│ │C7│ │C8│
├─────┤ ├─────┤ ├──┤ ├──┤ ├──┤ ├──┤
│xxxxx│ │xxxxx│ │  │ │  │ │  │ │  │
│xxxxx│ │xxxxx│ │  │ │  │ │  │ │  │
│xxxxx│ │xxxxx│ │  │ │  │ │  │ │  │
└─────┘ └─────┘ └──┘ └──┘ └──┘ └──┘
```

In general, this would be better however in this particular case, the selectivity of the first selection operator is extremely low, so such a merge of C_2, C_3 would actually *increase* the number of cache misses. The merge of C_1, C_5 however would decrease the number of cache misses by around 1, again since the selectivity is so low. So the optimal layout would be:

```
┌─────┐ ┌──┐ ┌──┐ ┌──┐ ┌──┐ ┌──┐ ┌──┐
│C1-C5│ │C2│ │C3│ │C4│ │C6│ │C7│ │C8│
├─────┤ ├──┤ ├──┤ ├──┤ ├──┤ ├──┤ ├──┤
│xxxxx│ │xx│ │xx│ │  │ │  │ │  │ │  │
│xxxxx│ │xx│ │xx│ │  │ │  │ │  │ │  │
│xxxxx│ │xx│ │xx│ │  │ │  │ │  │ │  │
└─────┘ └──┘ └──┘ └──┘ └──┘ └──┘ └──┘
```

Here we have around 2048 memory misses as before, since the selection operator has an extremely high selectivity. We save a few memory misses when accessing C_1-C_5 but this is mostly negligible. Similarly the cache utilization is only negligibly improved here.

Finally for S_3, the optimal layout is:

```
┌───────────────────────┐
│C1-C2-C3-C4-C5-C6-C7-C8│
├───────────────────────┤
│xxxxxxxxxxxxxxxxxxxxxxx│
│xxxxxxxxxxxxxxxxxxxxxxx│
│xxxxxxxxxxxxxxxxxxxxxxx│
└───────────────────────┘
```

We've already calculated the number of memory misses for this layout, 8192.

Notice that each row now has size 32 so each cache line fits two rows. Furthermore, the selection query has a very high selectivity and we need to output all of the row data at the end so this layout results in the fewest amount of cache misses.

Question 3 [40 pts]

There is a single table in our database. There are 8.192×10^6 (i.e., 8192000) rows in this table. There are four columns: A, B, C, D. Column A contains values between 0 and 100 with uniform distribution. Column B contains values -10^6 to 10^8 with uniform distribution. Column C contains values between 0 and 100 with uniform distribution. Column D contains values between 0 and 10^3 with uniform distribution. All the columns are of integer data type. An integer is 4 bytes.

Assume the following query:

```
select sum(A), avg(A), sum(B), sum(B*C), count(*)
from table
where A > 50 and B < 10^6 and D < 10^3
```

```
```

$\textbf{Q1.}$ **Write the code for select, fetch, average, sum, multiply, and count primitives that is necessary to execute the query above for a column-store engine. Write the code in C. [10 pts]**

For the following C code, we'll ignore degenerate edge cases such as passing in incorrect inputs to the operators or passing in invalid pointers, etc.

```c
selection *select(column *col, int* low, int* high) {
    selection *result = allocate_selection();

    for (size_t i = 0; i < col->size; ++i) {
        // Here the `compare(int* a, int* b)` returns a > b with special
        // cases for if `a` or `b` are `null`, e.g. `compare(null, 10)=1`
        if (compare(&col->data[i], low) && compare(high, &col->data[i]) {
            selection_push(result, i);
        }
    }

    return result;
}
```

```c
column *fetch(column *col, selection *sel) {
    column *result = allocate_column(sel->size);

    for (size_t i = 0; i < sel->size; ++i) {
        result_push(col->data[sel->data[i]]);
    }

    return result;
}
```

```c
float *average(column *col) {
    float result = 0.0;

    for (size_t i = 0; i < col->size; ++i) {
        result += ((float) col->data[i]) / ((float) col->size);
    }

    return result;
}
```

```c
long *sum(column *col) {
    long result = 0;
    for (size_t i = 0; i < col->size; ++i) {
        result += (long) col->data[i];
    }

    return result;
}
```

```
column *multiply(column *a, column *b) {
    assert(a->size == b->size);
    column *result = allocate_column(a->size);

    for (size_t i = 0; i < a->size; ++i) {
        result_push(a->data[i] * b->data[i]);
    }

    return result;
}
```

```
size_t count(selection *sel) {
    return sel->size;
}
```

$\textbf{Q2.}$ **Based on the data characteristics described above, what would your optimal query plan be in terms of the order of operations? Explain in detail. [7.5 pts]**

To minimize data movement, we'll order the three selection operators in order of increasing selectivity. We can do this since the data is independently distributed among the three columns and each column has a uniform distribution. Observe that $A > 50$ has a selectivity of $\approx 50%$, $B < 10^6$ has a selectivity of $\approx 2%$ and $D < 10^3$ has a selectivity of $\approx 100%$. So the optimal query plan which uses the given operators is

```
s1 = select(B, less_than(10^6))
s2 = select(A, s1, greater_than(50))
s3 = select(D, s2, less_than(10^3))

a = fetch(A, s3)
b = fetch(B, s3)
c = fetch(C, s3)

aa = average(a)
sa = sum(a)

sb = sum(b)
m = multiply(b, c)

sm = sum(m)
```

```
co = count(s3)

output(sa, aa, sb, sm, co)
```

This is more optimal than a different ordering of the select operators since if high selectivity operators are used first, we must load most of the first and second columns as opposed to loading some of the first and even less of the second.

We also order the aggregation operators in a way that improves cache performance, for example all of the A related operators are next to each other.

$\textbf{Q3.}$ **Calculate the cost of your optimal query plan in $\textbf{Q2}$ in terms of number of CPU cache misses (ignore the TLB cost). All your data resides in main memory and there is no disk access. For each page read/write, one cache miss is incurred. Assume that every time a tuple is randomly accessed (i.e., read/written non-sequentially), the whole page is loaded into the cache, i.e., one cache miss is incurred. Assume that caches are smart enough not to evict a page that is actively being used. Assume branches that have less than $10%$ or greater than $90%$ selectivity are predictable and cost zero. Unpredictable branches cause branch mispredictions and assume a branch misprediction costs three cache misses. [10 pts]**

We assume that the cache is empty at the start of the query execution, and is not being filled by any other process while the query is running. Also assume that the cache line size is 64 bytes.

Start with the first line:

```
s1 = select(B, less_than(10^6))
```

This will load in the entirety of B into the cache consisting of $4\times 8192\times 1000$ bytes or 512000 cache lines. In line with this, There are only 2000 cache lines in the cache, so we have $256 \times 2000=512000$ cache misses from accessing B. Also for about $2%$ of the entries in the column, we must add it to the result. This means we get an additional 10240 cache misses, and in total 522240.

Since the if statement is hit only $2%$ of the time, we don't care about the branch misprediction cost.

Next, we have:

```
s2 = select(A, s1, greater_than(50))
```

We can't assume that s_1 is still loaded in the cache, so we'll have 10240 cache misses. Each of these corresponds to 16 selection indices. In the worst case scenario when all indices correspond to a different page, we get $10240 * 16$ cache misses. On top of that have a $50%$ selectivity if statement for each of these, so we add another $10240 * 8 * 3$ cache misses to account for this. Lastly, we add 10240×0.50 cache misses because of the selection array s_2. So this line of code adds 419840 cache misses.

The last line of the selection is:

```
s3 = select(D, s2, less_than(10^3))
```

As before, we can't assume that s_2 is loaded in the cache, so worst case we'll have 5120 cache misses to load s_2. Each of these corresponds to 16 selection indices, and in the worst case when all the indices correspond to a different page we get $5120 * 16$ cache misses. The branch misprediction cost can be ignored. Finally, we also have 5120 cache misses corresponding to s_3. So this line of code adds 92160 cache misses.

So the selection operators incurred a cost of around 1000000 cache misses. (In the worst case) s_3 also consists of 5120 selection indices, which in the worst case are spread out evenly among the rows.

Next we have the aggregate operations:

```
a = fetch(A, s3)
b = fetch(B, s3)
c = fetch(C, s3)

aa = average(a)
sa = sum(a)
sb = sum(b)
m = multiply(b, c)
sm = sum(m)

co = count(s3)
```

Using a similar analysis as before, in total we incur and extra 291840 cache misses. So in total we have around 1.3×10^6 cache misses, which corresponds to 83 megabytes of data movement between main memory and the cache.

$\textbf{Q4.}$ **Consider the code below that implements the same query but consolidated in a single function. Compute the cost of this new function and compare to the cost you calculated in Q3 in terms of number of cache misses (ignore the TLB cost). Assume that the predicates in the if statement below are executed from left-to-right, and the if-statement execution stops as soon as one of the predicates (e.g., $B[i] < 10^6$) is evaluated to false. Assume branches that have less than 10% or greater than 90% selectivity are predictable and cost zero. Unpredictable branches cause branch misprediction and assume a branch misprediction costs three cache misses. [7.5 pts]**

```
int_tuple execute_my_beautiful_query(num_row) {
    acc0=acc1=acc2=count=0;
    for(i=0; i<num_row; i++) {
        if(B[i] < 10^6 && A[i] > 50 && D[i] < 10^3) {
            acc0 += A[i];
            acc1 += B[i];
            acc2 += (B[i]*C[i]);
```

```
                count += 1;
            }
        }
        return (acc0, acc0/count, acc1, acc2, count);
    }
```

We have 8192×1000 rows, and each cache line consists of 16 lines so we have 512000 cache lines per column. Since we're accessing all of the columns with the same index, we only care about blocks of 16 indices. Regardless of the result of the if statement, we always read B, so we have 512000 cache misses. $B[i] < 10^6$ has a selectivity of 0.02, so the probability that there will be a hit in a block of 16 indices is $1-(1-0.02)^{16}\approx 0.28$. So next we load $A[i]$ on average 0.28×512000 times. This has selectivity 0.50 so we load $D[i]$ on average 0.14×512000 times. The last condition has a $100%$ selectivity, so the if statement has a selectivity 0.01.

In the body if the if statement, A and B are already loaded by the if statement, we only need to load 0.14×512000 cache lines of C. So the total number cache misses is 798720.

We still need to factor in the if statement cost, since every `&&` is essentially an if statement. The selectivity of the first condition is 0.02 which is negligible, but the second condition is 0.50 so we have $0.02 \times 8192000 \times 0.5$ branch mispredictions which adds $0.02 \times 8192000 \times 0.5 \times 3$ cache misses to the total.

So the total cost is $1044480 \approx 1.0\times 10^6$ cache misses.

$\textbf{Q5.}$ **Implement the branchless version of the code in Q4. Estimate the cost of the branchless version in terms of number of cache misses (ignore the TLB cost). Which version of the code is costlier: branchless or branched? Why? [5 pts]**

```
int_tuple execute_my_beautiful_query(num_row) {
    acc0=acc1=acc2=count=0
    for(i=0; i<num_row; i++) {
        int condition = B[i] < 10^6 & A[i] > 50 & D[i] < 10^3;
        acc0 += condition * A[i];
        acc1 += condition * B[i];
        acc2 += condition * B[i]*C[i];
        count += condition;
    }
    return (acc0, acc0/count, acc1, acc2, count);
}
```

Since there are 512000 cache lines of data in each column, and we're accessing each column in the loop, we get 4×512000 cache misses which is about 2.0×10^6 cache misses.
