Cairo University - Faculty of Engineering

Computer Engineering Department

Advanced Database Systems

Project Phase Two

Mohamed Shawky Zaky
SEC:2, BN:15

Remonda Talaat Eskarous
SEC:1, BN:19

Mohamed Ahmed Mohamed Ahmed
SEC:2, BN:10

Mohamed Ramzy Helmy
SEC:2, BN:13

Contents

List of Figures

1 Query Statistics

In this section, we show both **query trees** and **MySQL execution plan** for the *non-optimized* and *optimized* version of each query. Moreover, we provide *parallel query processing reports* for each query. This illustrates the optimization of the query statements that might include *schema, query rewriting, semantic and statistical heuristics*.

Note that, the cost estimates in *MySQL* execution plan can be inaccurate, due to the following :

1. The unit of the cost is an abstract number that not necessarily relate to the real execution. It's just used as a heuristic to order certain operations in the query execution.

2. *MySQL* estimator considers equal costs for both *CPU* and *IO* operations, which isn't valid, due to *modern processors' speeds*.

3. *MySQL* estimator doesn't consider database indexes.

Moreover, the provided queries can be run in **parallel** given a multiprocessor *(multi-core processor)* device and a *DBMS* that supports parallelism. Unfortunately, we use *MySQL*, which doesn't support parallel execution plans. So, we provide the *parallel query processing reports* through theoretical analysis.

1.1 Query 1

1.1.1 Execution Plan Before Optimization

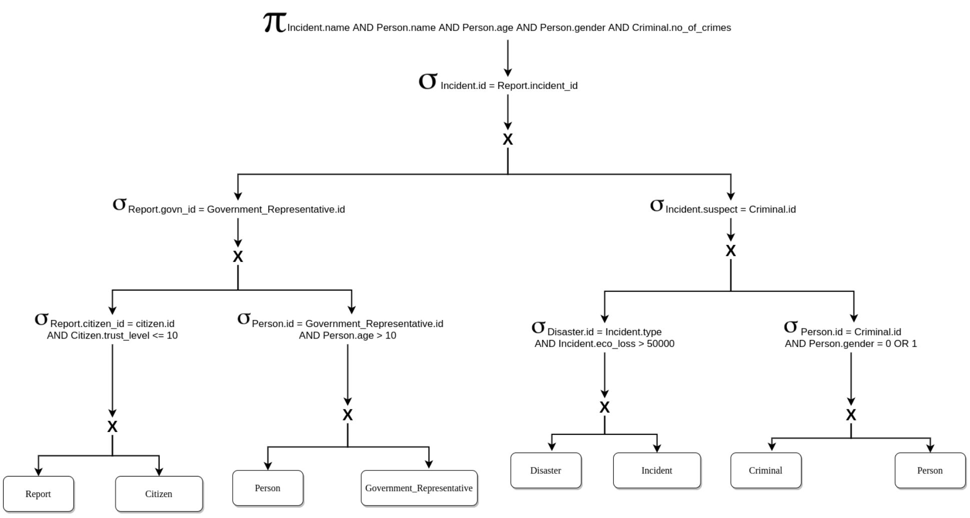

Figure 1: Initial version of query tree for non-optimized query 1

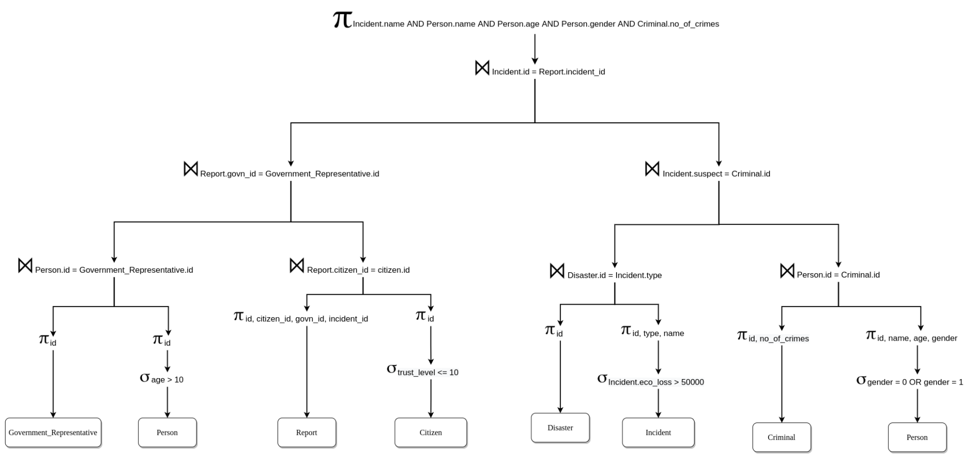

Figure 2: Final version of query tree for non-optimized query 1

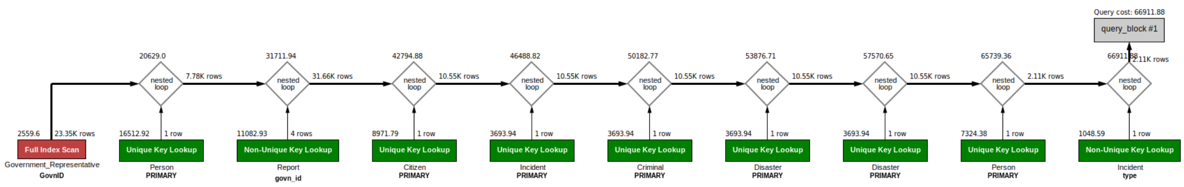

Figure 3: Visual execution plan for non-optimized query 1

1.1.2 Execution Plan After Optimization

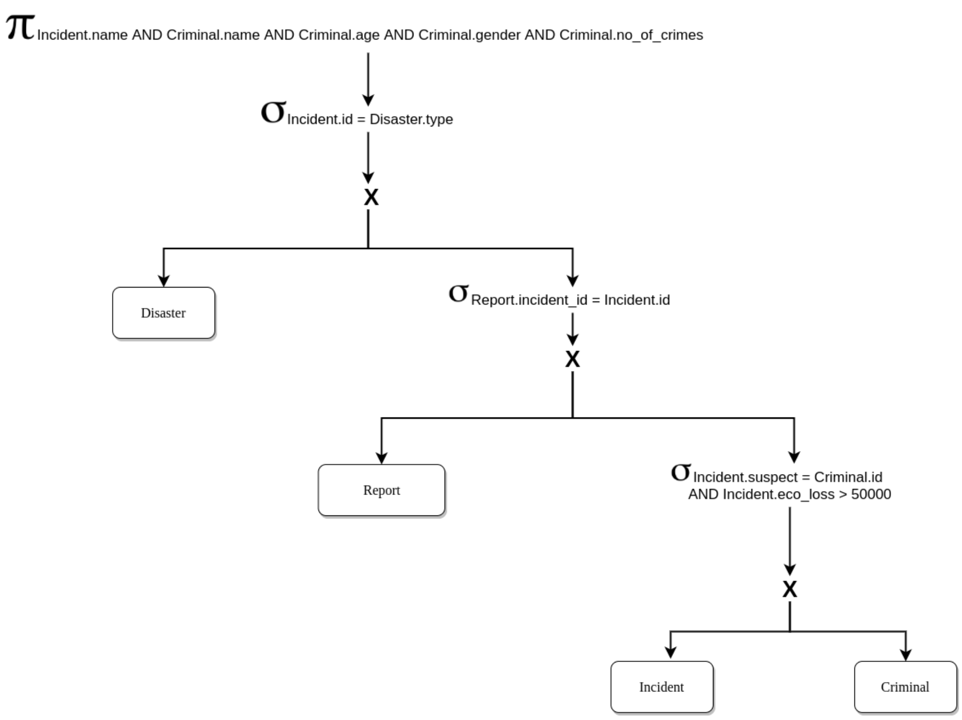

Figure 4: Initial version of query tree for optimized query 1

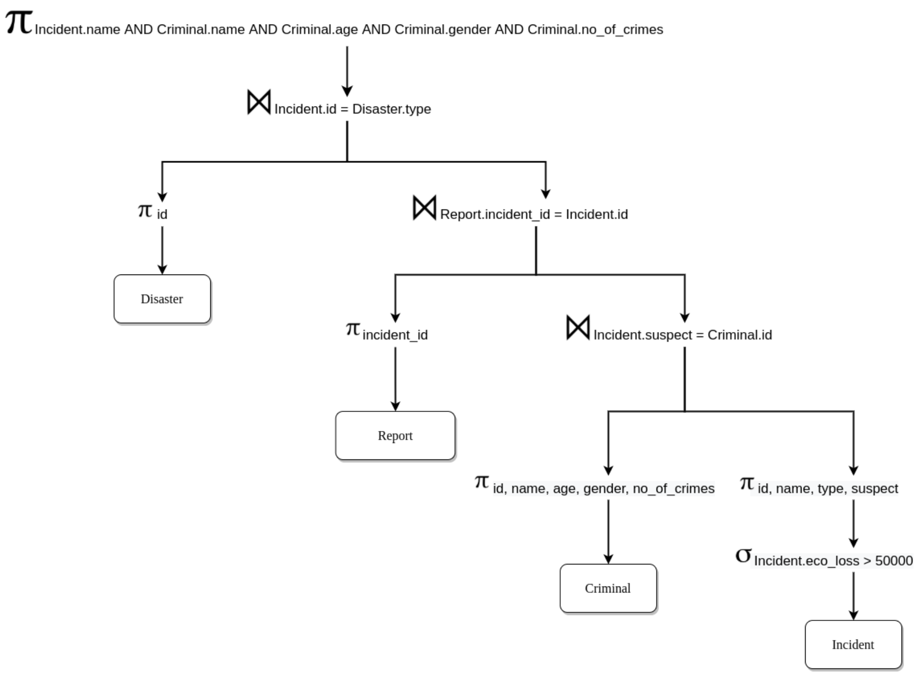

Figure 5: Final version of query tree for optimized query 1

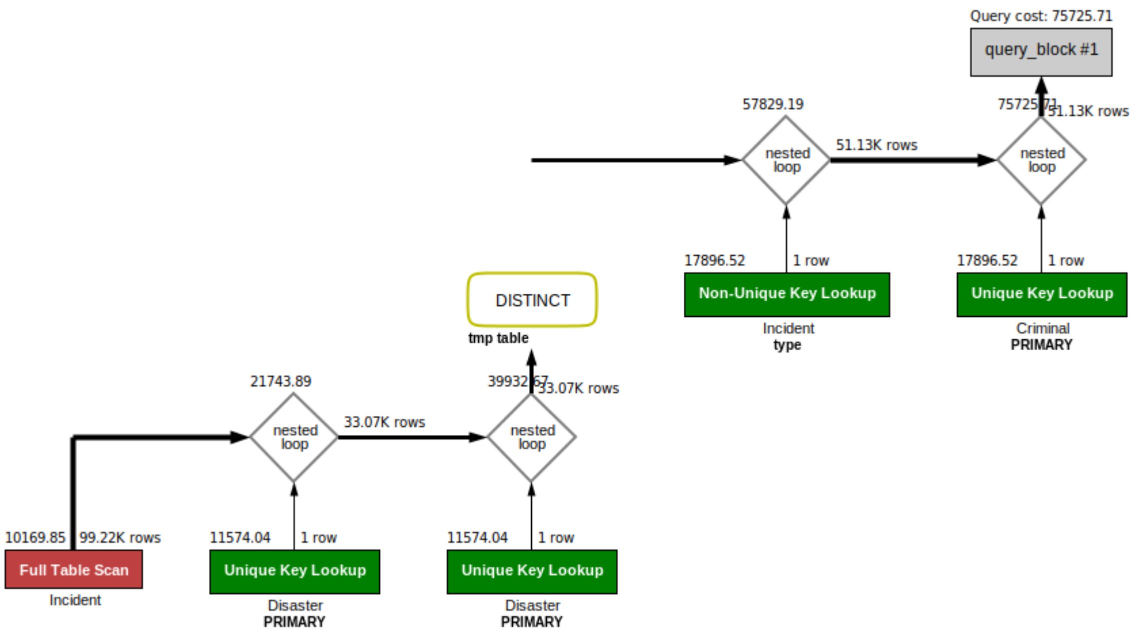

Figure 6: Visual execution plan for optimized query 1

1.1.3 Parallel Query Execution

We can do the following analysis theoretically :

- The **incident** table is fully scanned through a single worker (thread). Since we are using a *4-core* processor, This scan can be run on 4 threads. The results from 4 streams are, then, gathered into a single stream. This can reduce the full scan time in the previous execution plan to *quarter*.

1.2 Query 2

1.2.1 Execution Plan Before Optimization

Note that, due to the huge execution time of the query before adding the indexes, we can't extract the actual non-optimized visual execution plan. Therefore, we included the visual execution plan after *index optimization*.

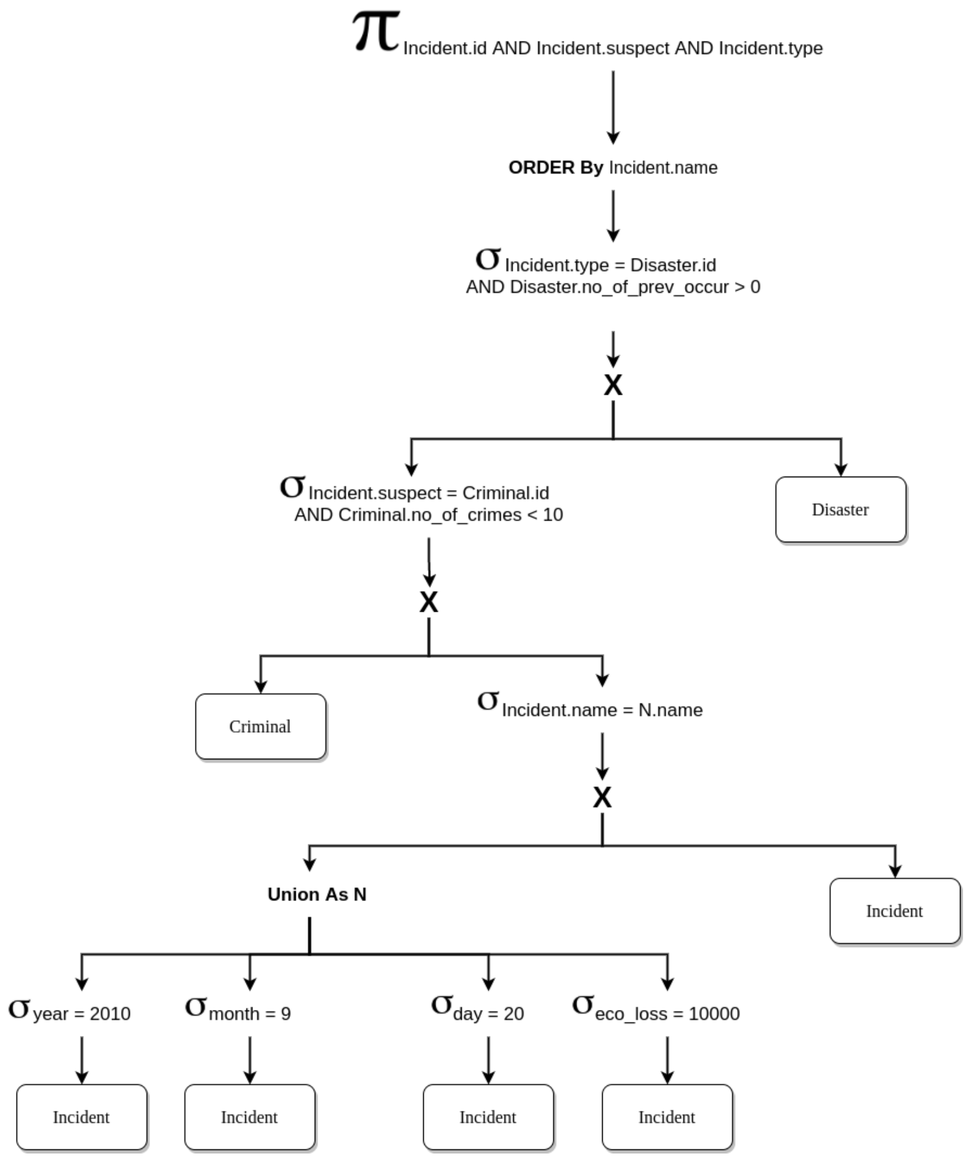

Figure 7: Initial version of query tree for non-optimized query 2

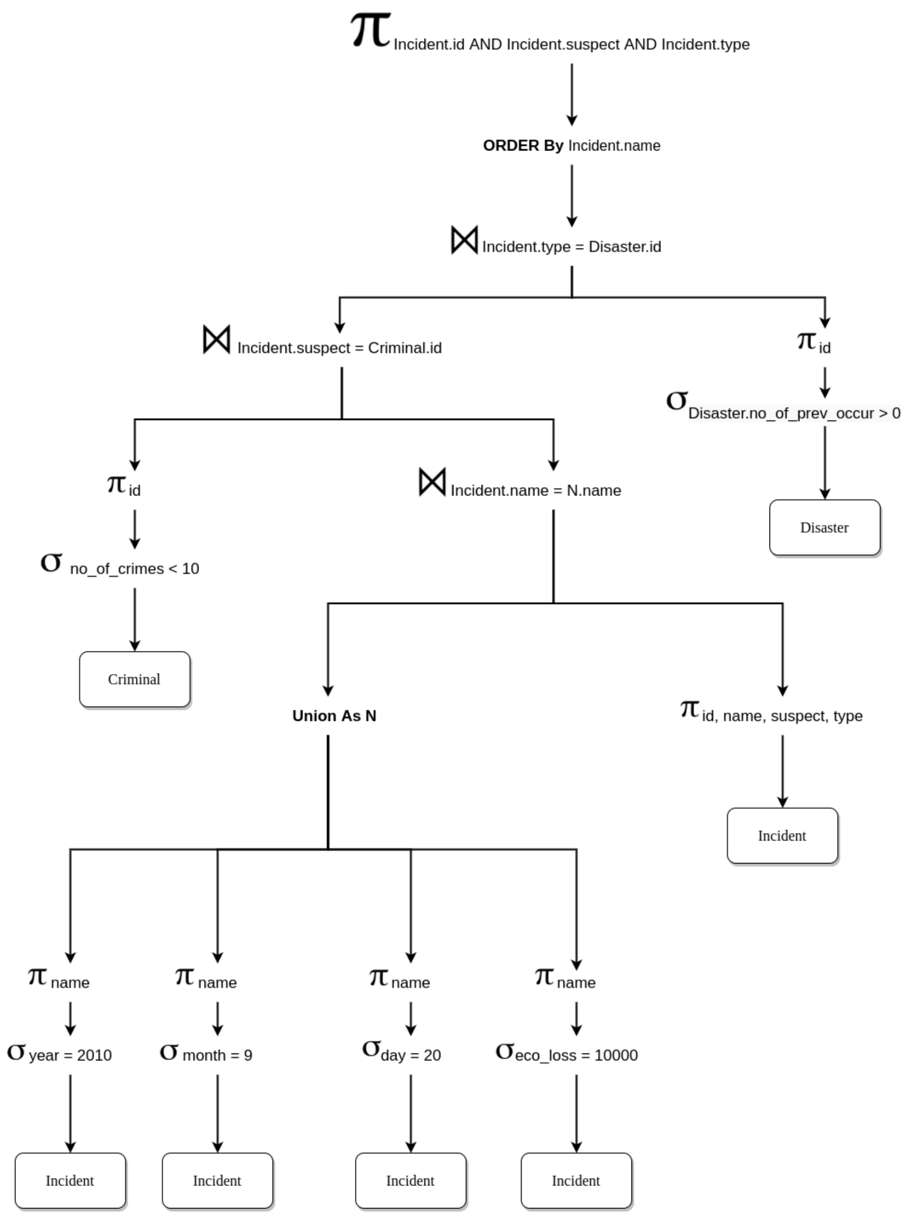

Figure 8: Final version of query tree for non-optimized query 2

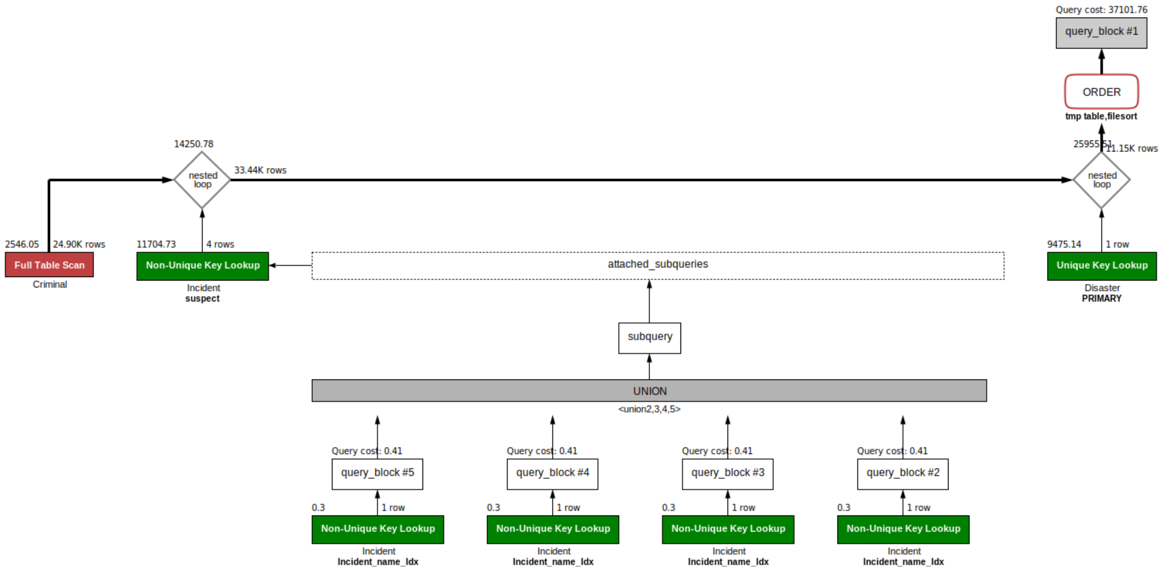

Figure 9: Visual execution plan for non-optimized query 2

1.2.2 Execution Plan After Optimization

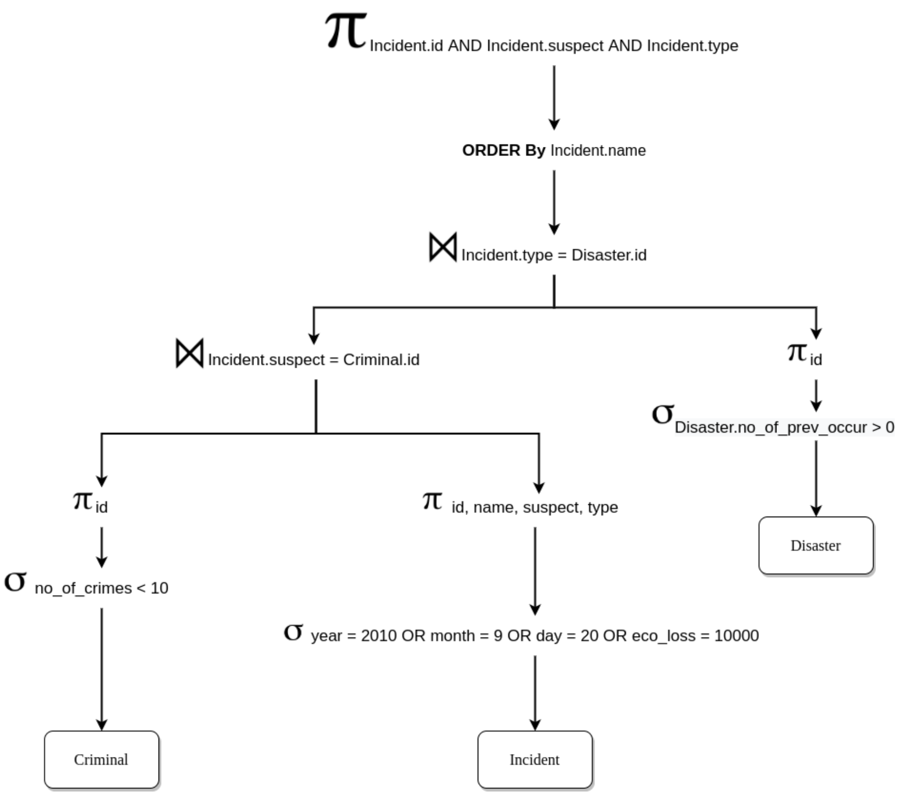

Figure 10: Initial version of query tree for optimized query 2

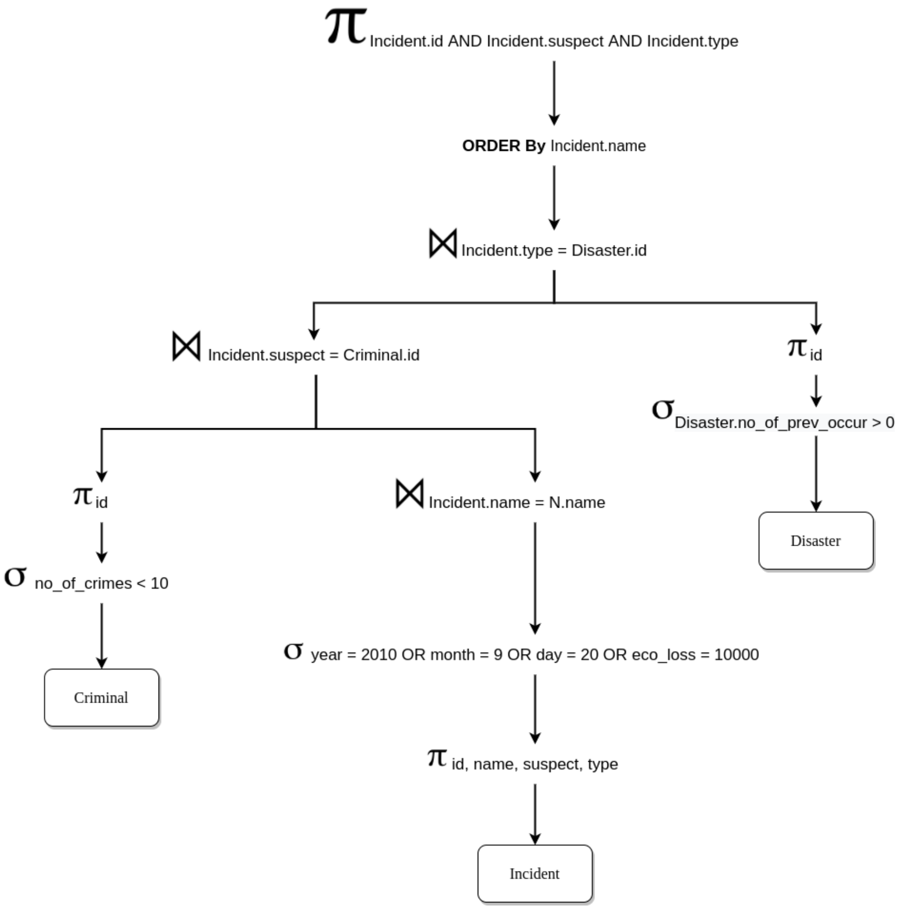

Figure 11: Final version of query tree for optimized query 2

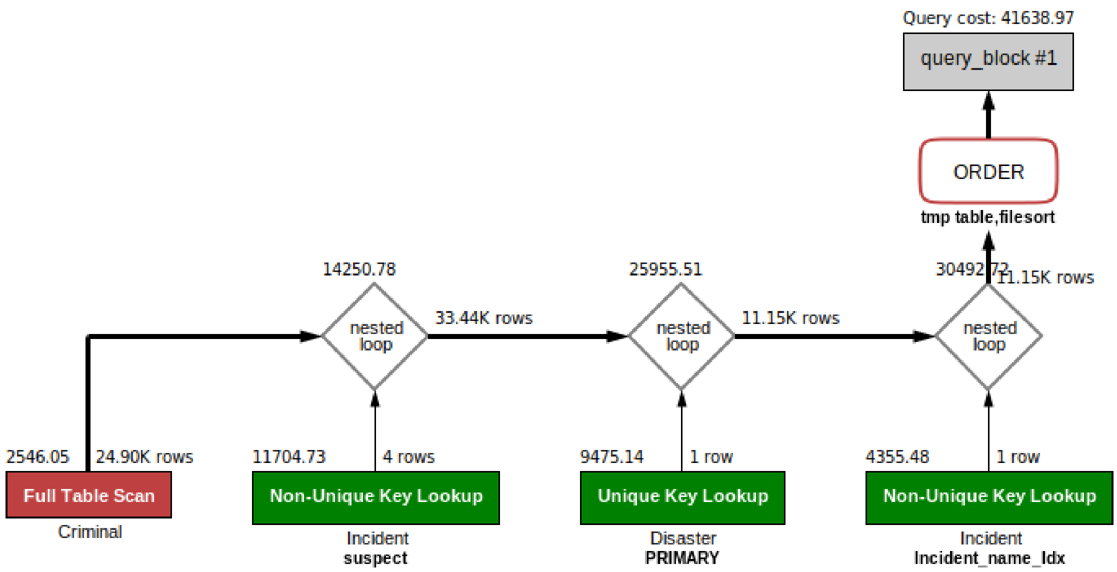

Figure 12: Visual execution plan for optimized query 2

1.2.3 Parallel Query Execution

We can do the following analysis theoretically :

- According to the previous *execution plan*, the *criminal* table is fully scanned at the beginning. This operation is done on a single thread. However, it can be executed in parallel on 4 threads, which can reduce time to *quarter*. The 4 previous streams can be gathered in a single stream.
- Also, the resulting temporary table is sorted in a single thread. However, it can be done on 4 threads and then gathered to significantly reduce the costy time of sort.
- Also note that using *or* operation is the fastest on a single thread, however using *unions* instead can provide more potential for parallel execution. Each **union** can be executed on a thread and then gathered to reduce time to *quarter*.

1.3 Query 3

1.3.1 Execution Plan Before Optimization

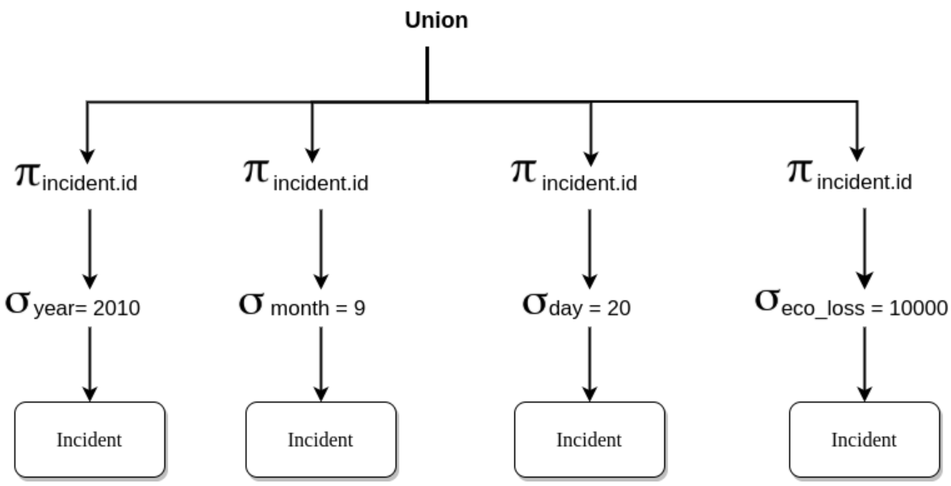

Figure 13: Query tree for non-optimized query 3

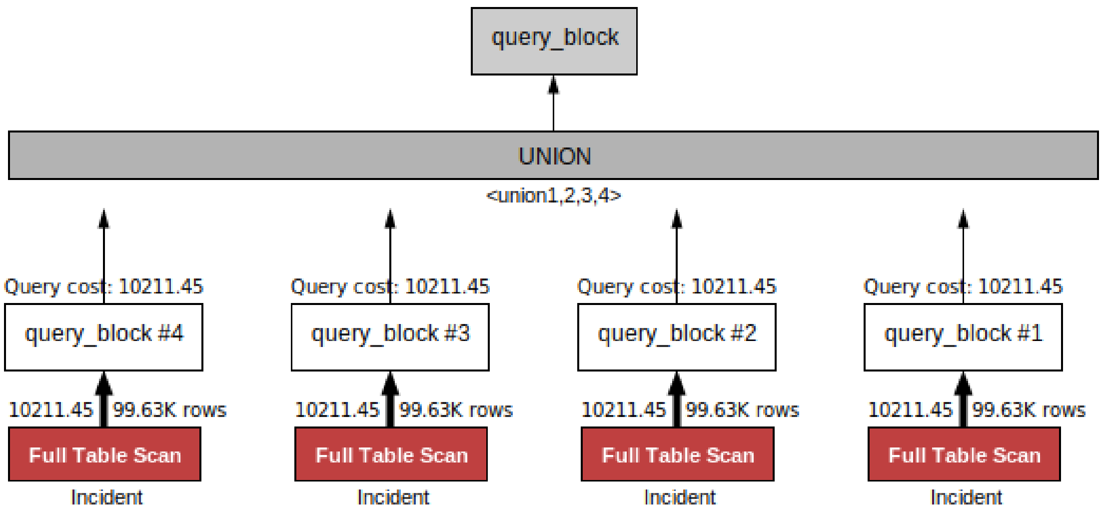

Figure 14: Visual execution plan for non-optimized query 3

1.3.2 Execution Plan After Optimization

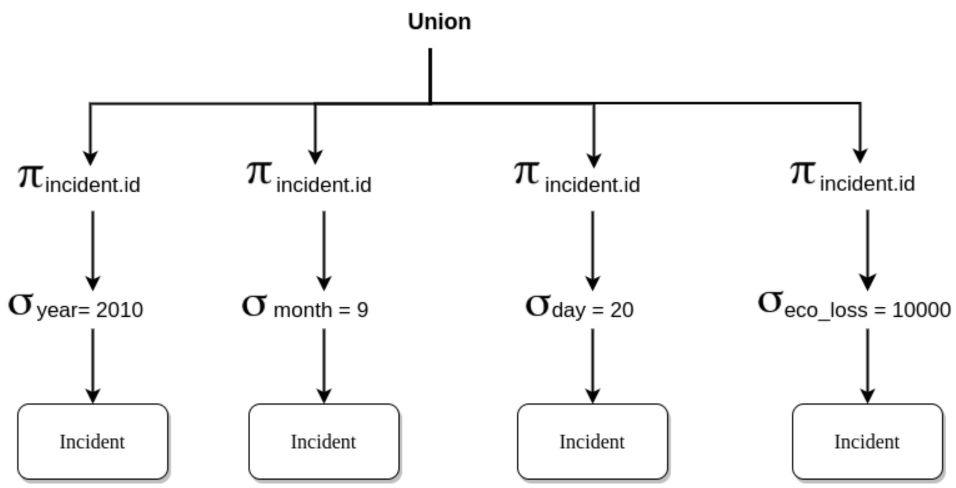

Figure 15: Query tree for optimized query 3

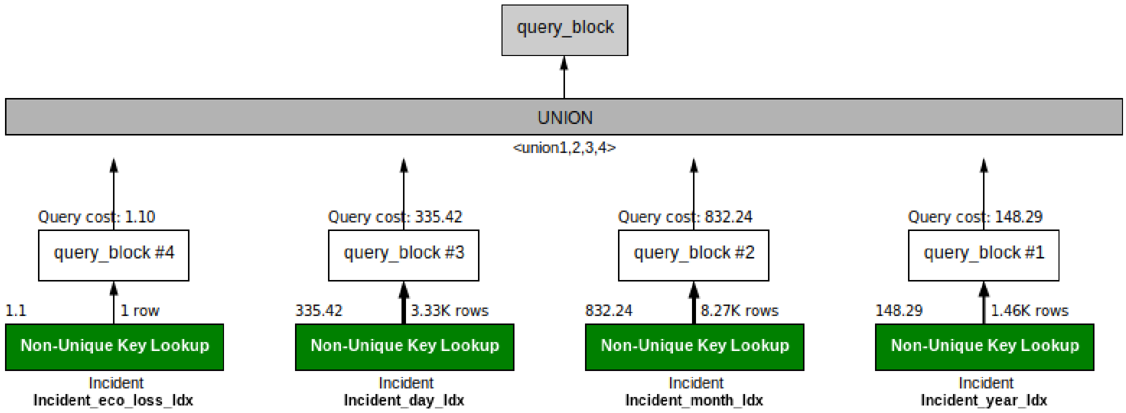

Figure 16: Visual execution plan for optimized query 3

1.3.3 Parallel Query Execution

We can do the following analysis theoretically :

- From the previous *execution plan*, it's obvious that the query can be divided on 4 threads, one for each *union* operation. The results are, then, gathered from 4 streams, which allows for time reduction up to *quarter*.

1.4 Query 4

1.4.1 Execution Plan Before Optimization

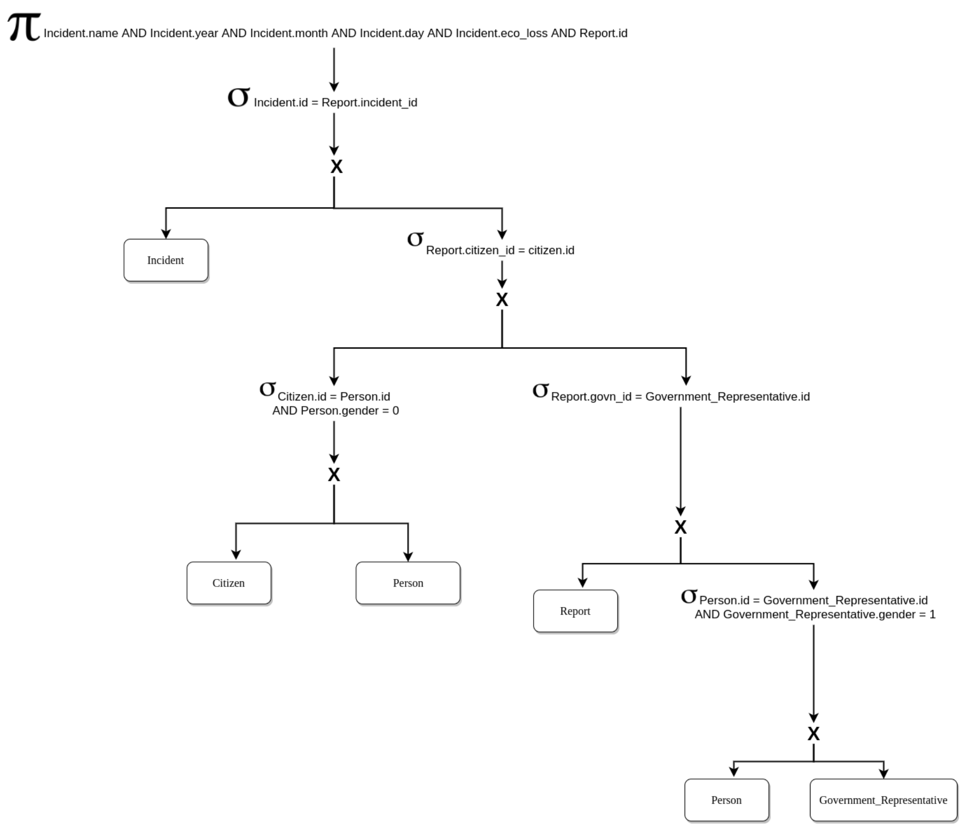

Figure 17: Initial version of query tree for non-optimized query 4

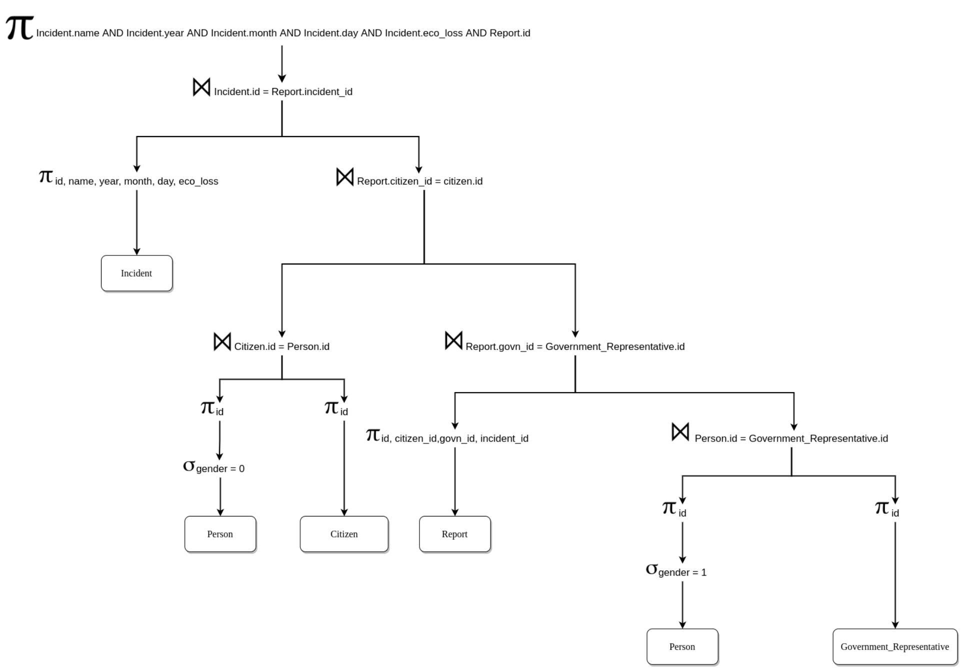

Figure 18: Final version of query tree for non-optimized query 4

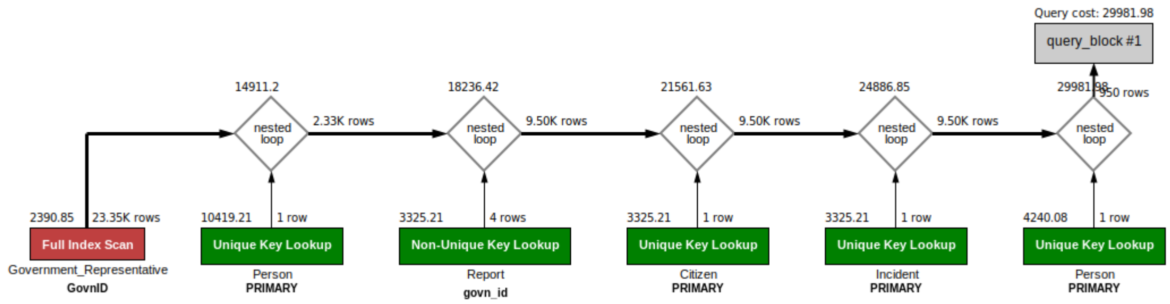

Figure 19: Visual execution plan for non-optimized query 4

1.4.2 Execution Plan After Optimization

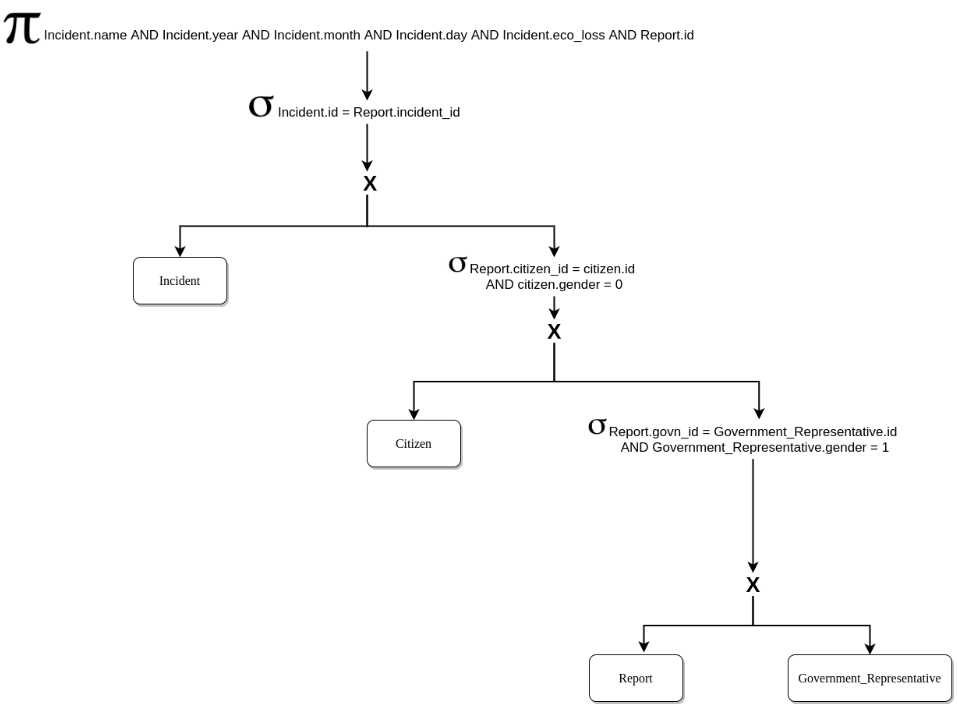

Figure 20: Initial version of query tree for optimized query 4

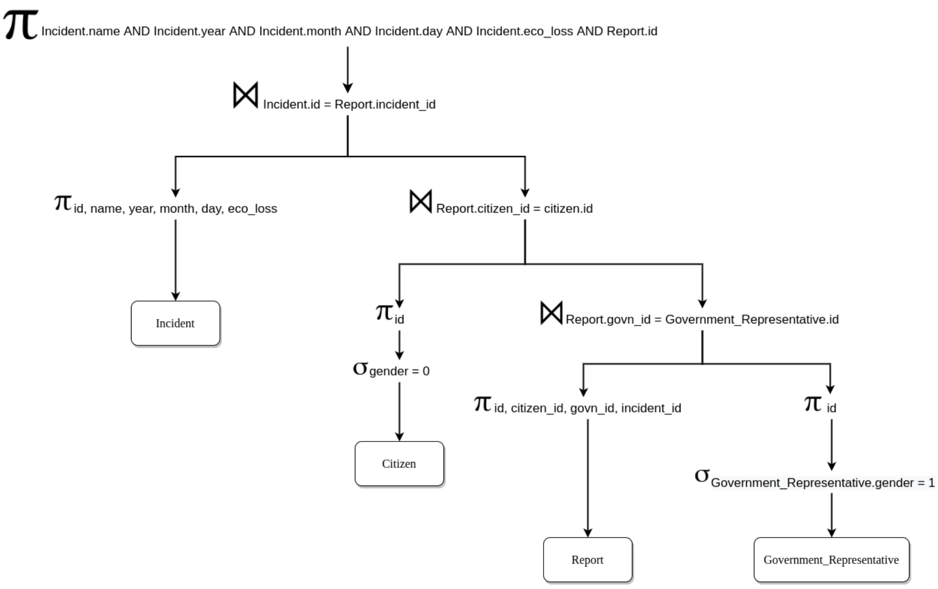

Figure 21: Final version of query tree for optimized query 4

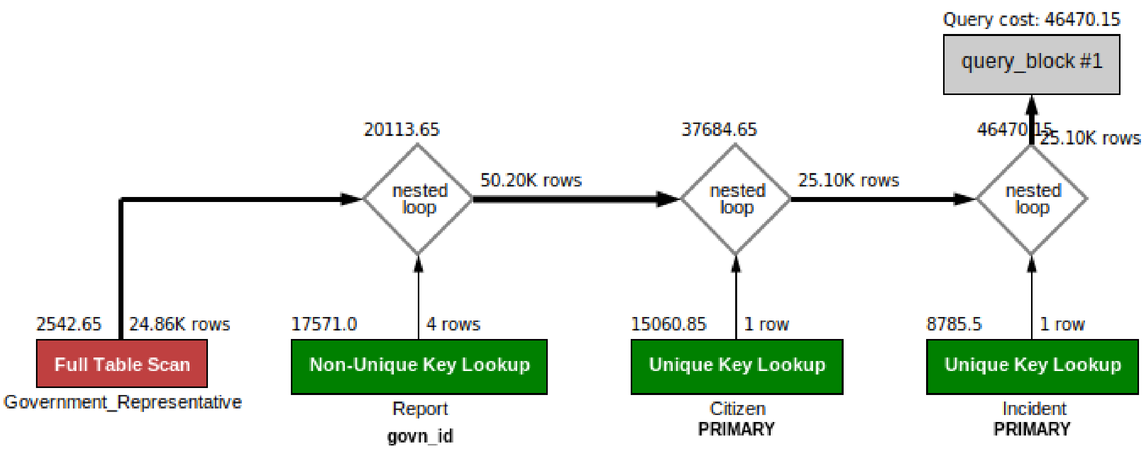

Figure 22: Visual execution plan for optimized query 4

1.4.3 Parallel Query Execution

We can do the following analysis theoretically :

- The **Government_Representative** table is fully scanned through a single worker (thread). Since we are using a *4-core* processor, This scan can be run on 4 threads. The results from 4 streams are, then, gathered into a single stream. This can reduce the full scan time in the previous execution plan to *quarter*.

1.5 Query 5

1.5.1 Execution Plan Before Optimization

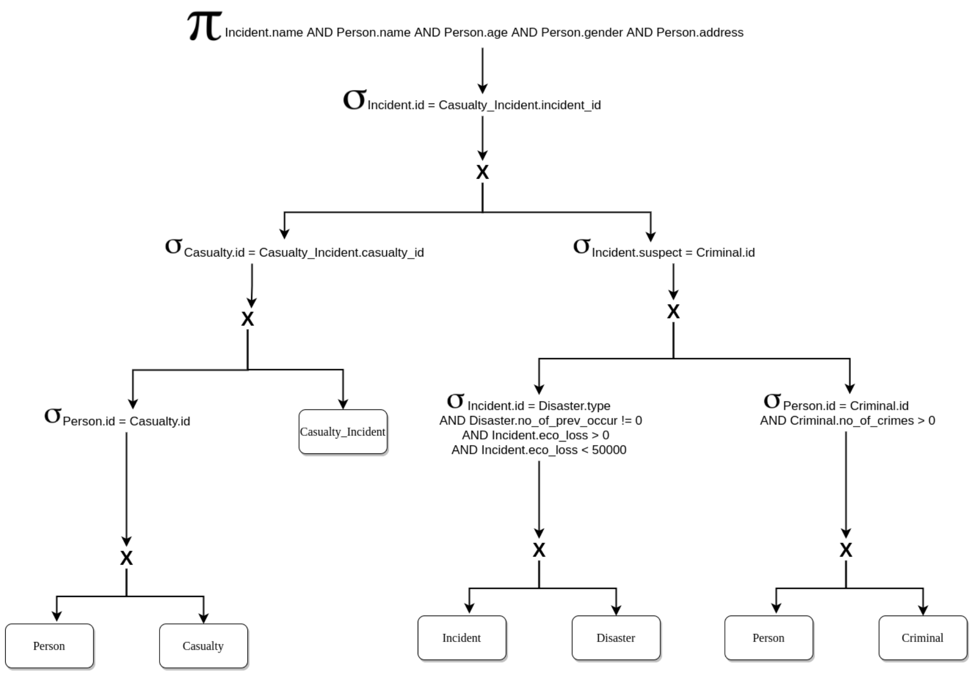

Figure 23: Initial version of query tree for non-optimized query 5

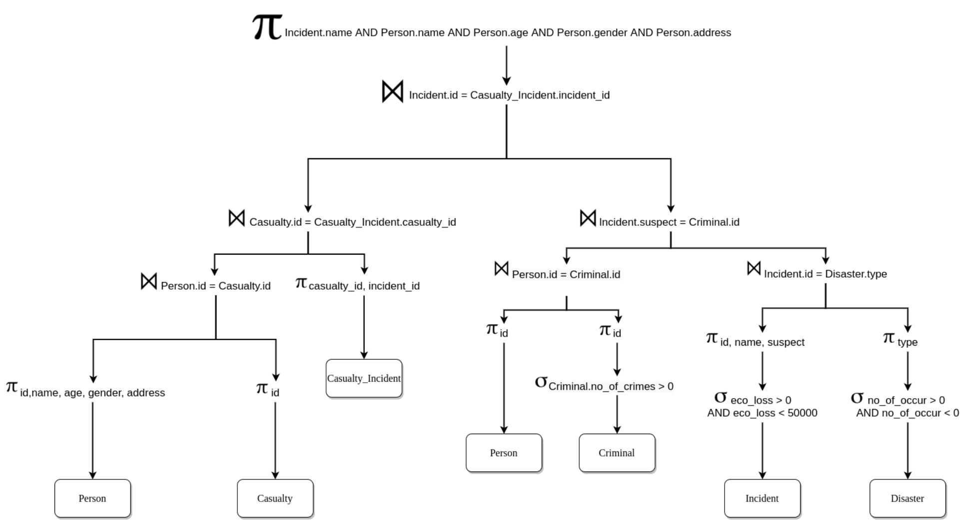

Figure 24: Final version of query tree for non-optimized query 5

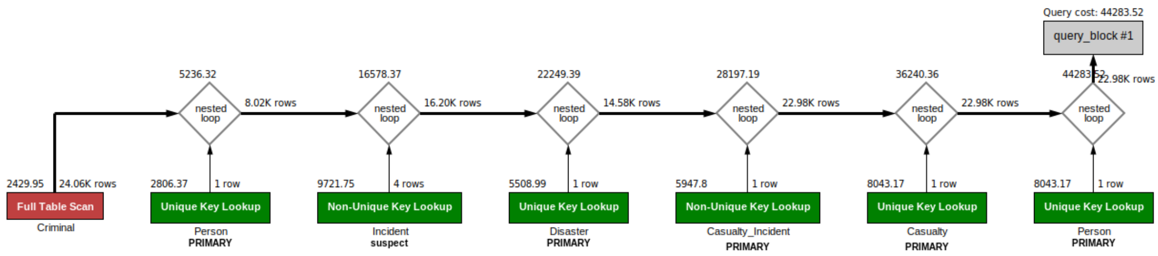

Figure 25: Visual execution plan for non-optimized query 5

1.5.2 Execution Plan After Optimization

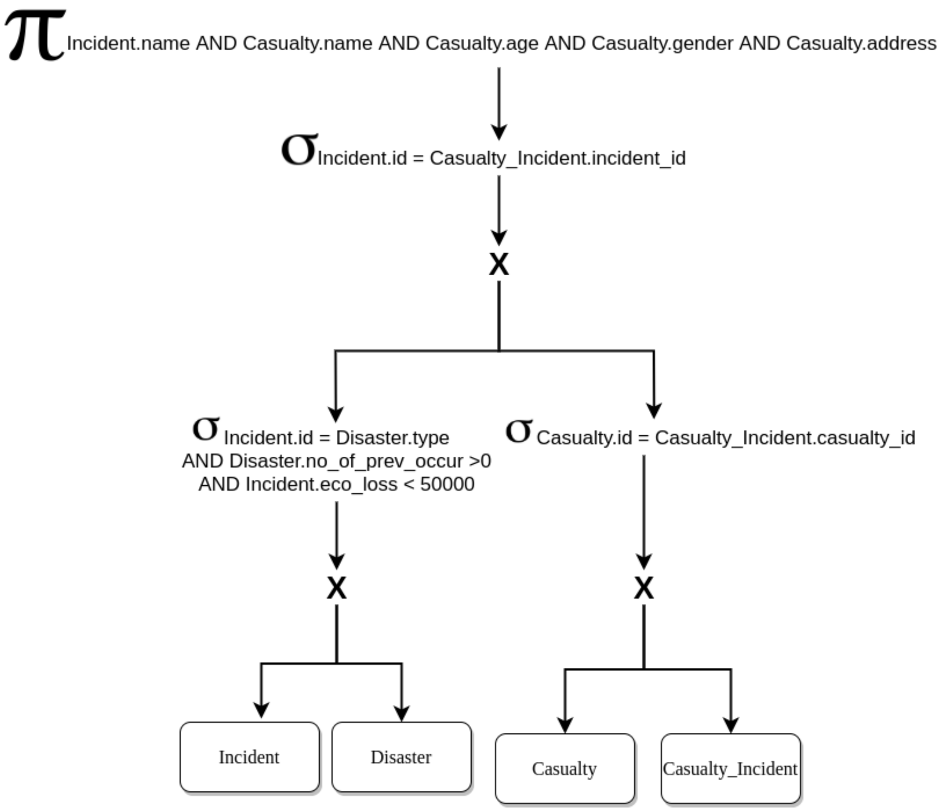

Figure 26: Initial version of query tree for optimized query 5

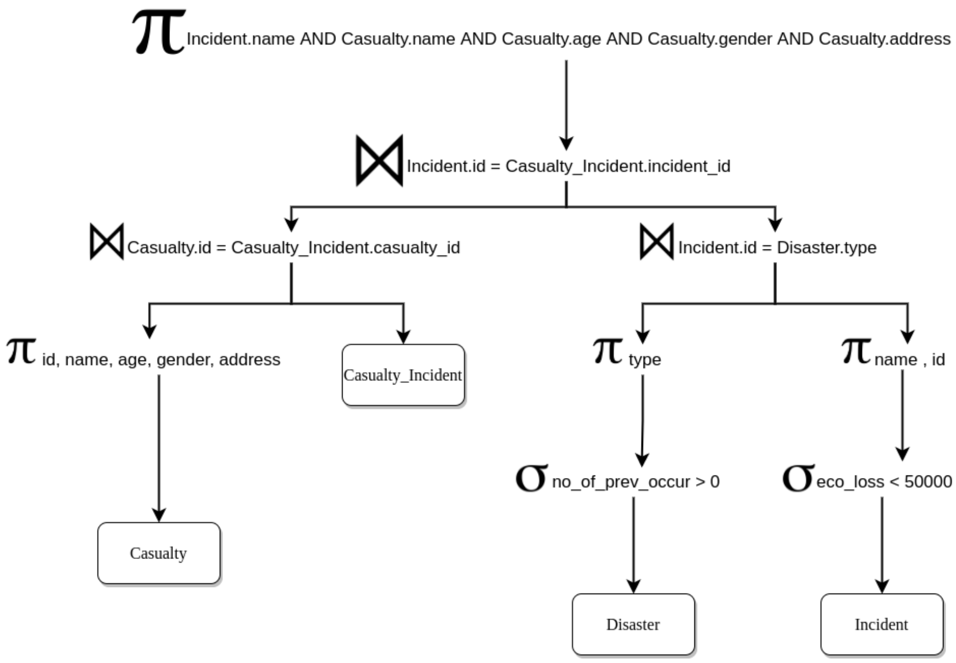

Figure 27: Final version of query tree for optimized query 5

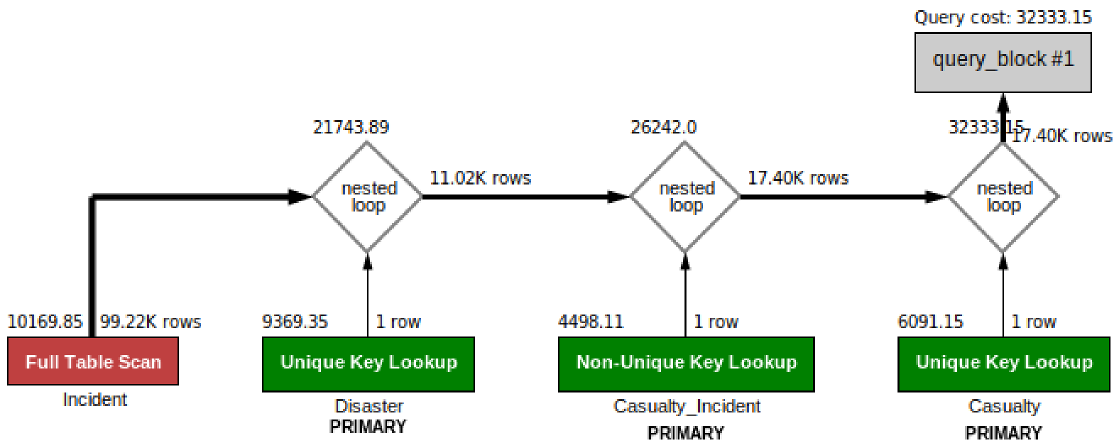

Figure 28: Visual execution plan for optimized query 5

1.5.3 Parallel Query Execution

We can do the following analysis theoretically :

- The **incident** table is fully scanned through a single worker (thread). We can see from the estimated cost that this takes up most of the query time. Since we are using a *4-core* processor, This scan can be run on 4 threads. The results from 4 streams are, then, gathered into a single stream. This can reduce the full scan time in the previous execution plan to *quarter*. This will greatly affect the query execution time.

2 Optimization Details

In this section, we show the optimization details done through this work. We discuss the statistics of the new database and the schema changes. Moreover, other optimization techniques related to query, indexes and memory are discussed, as well.

2.1 New Database Statistics

In this subsection, we show the new database statistics after optimization. The record count is extracted from the database filled with 100000 records per table. Other filling sizes are considered through the analysis like 10000 and 1000000.

Table Name	Row Count	Main Key	Indexes	FK
Disaster	100000	YES	4	2
Causes	100000	YES	1	1
Precautions	100000	YES	1	1
Incident	100000	YES	4	3
Descriptions	100000	YES	1	1
Casualty	25000	YES	1	0
Government_Representative	25000	YES	1	0
Govn_Rep_Credentials	25000	YES	1	1
Citizen	25000	YES	1	0
Citizen_Credentials	25000	YES	1	1
Criminal	25000	YES	1	0
Report	100000	YES	5	4
Report_Content	100000	YES	1	1
Casualty_Incident	100000	YES *(Composite)*	3	2

Table Name	Identity Column	Max Row Size (Bytes)
Disaster	YES	52
Causes	YES	65538
Precautions	YES	65538
Incident	YES	120
Descriptions	YES	65538
Casualty	YES	105
Government_Representative	YES	116
Govn_Rep_Credentials	YES	103
Citizen	YES	116
Citizen_Credentials	YES	103
Criminal	YES	106
Report	YES	23
Report_Content	YES	65538
Casualty_Incident	NO	6

2.2 Schema Optimization

The following database schema shows the optimizations over the old schema. The schema optimizations can be summarized as follows :

1. **Denormalization** of *Person* table with all persons types. This is mainly because no duplicates between persons types. For example, no person can be a casualty and a criminal at the same time. So, in order to avoid redundant joins, the *Person* relation is merged into the four child relations.

2. **Normalization** *(Vertical Partitioning)* of variable-size data. The access of fixed-size records is faster than that of variable-size data, as the *DBMS* don't require to pre-calculate the record size. That's why, the variable-size data like *text*, *usernames* and *passwords* are separated into separate relations. One more reason for doing so is that the data in these fields aren't accessed frequently, so they better be separate from other frequently-accessed data.

3. **Minimization** of the data types based on semantic and statistical *heuristics*. The data types of different fields are reduced to the minimum possible size, in order to ease their read and write to the disk. For example, all *primary keys* are reduced to `MEDIUMINT`, because the table size is at most 1000000. Moreover, Some fields that are just limited to range from 1 to 10 are reduced to 4 bits instead of `INT`.

4. **Conversion** of variable-size data to fixed-size data. As the fixed-size record are faster to access and transfer, so each *VarChar* is converted into *Char*. This, however, can increase the storage space, as *Char* allocates the target bytes whatever they are all used or not.

5. **Usage** of `NOT NULL`, whenever possible. If the field isn't marked as `NOT NULL`, it allocates extra bits to check whether the field is *NULL* or not.

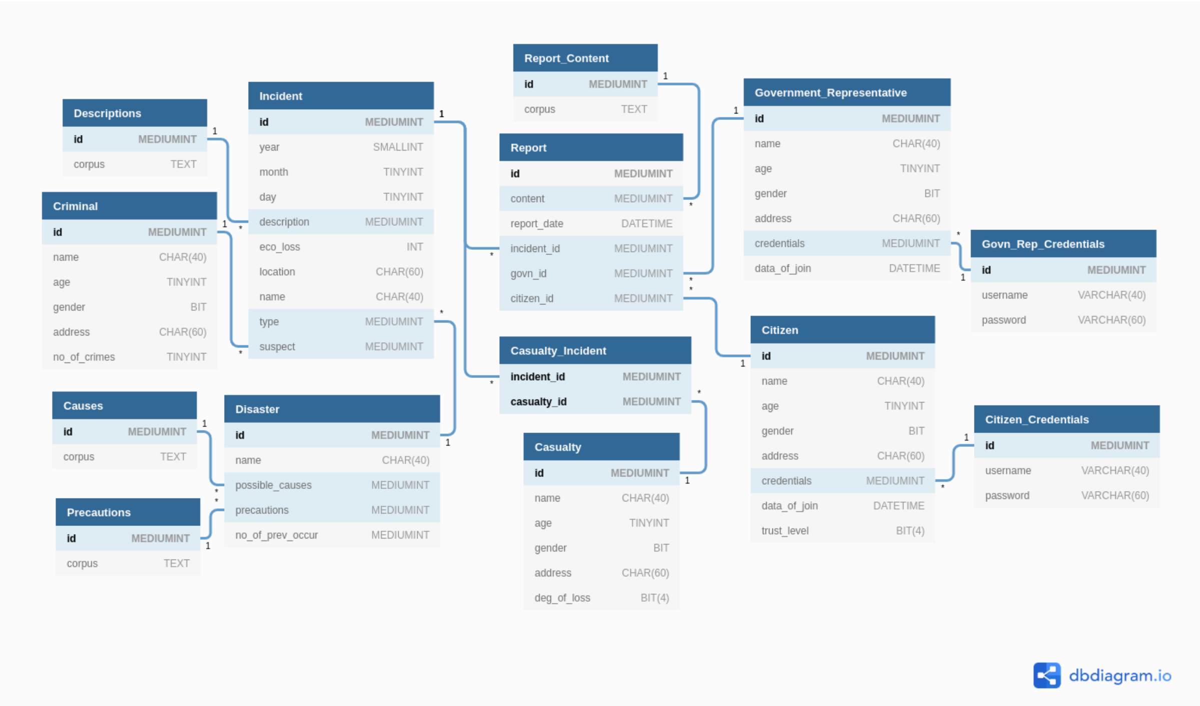

Figure 29: New Optimized Database Schema.

2.3 Memory Optimization

In *MySQL*, the memory optimization can be done through changing the memory system variables of the *DBMS* given the same hardware. The system variables can include `innodb_buffer_pool_size`, `innodb_buffer_pool_chunk_size` and *innodb_buffer_pool_instances*. We can, also, switch between the two storage engines of *MySQL*, which are `INNODB` and `MyISAM`.

We tried both storage engines and discovered that using `INNODB` gives much better performance based on our database schema. Moreover, we tried to optimize most of the system variables, however some changes are **prohibited** by *MySQL* and other changes don't affect the performance at all. The only change that results in significant improvement is increasing `innodb_buffer_pool_size`.

So, we can summarize our *memory optimization* as follows :

- Usage of `INNODB` as a storage engine, which is much faster.
- Increasing `INNODB`'s buffer pool size, which is responsible for the amount of memory allocated for *DBMS* operations, from *32MB* to *4GB*.

2.4 Index Tuning

2.5 Query Optimization

2.5.1 Query 1

2.5.2 Query 2

2.5.3 Query 3

2.5.4 Query 4

2.5.5 Query 5

3 Validation Details

3.1 Time and Space Analysis

In this subsection, we evaluate both time and space improvements of each optimization on each query. We consider both before and after *disk cache*. Moreover, the space improvement is considering the **total size** of the transferred tables between memory and disk. Execution time is measured in *seconds*.

Query 1	Before Cache			After Cache		
	Time	Time %	Space %	Time	Time %	Space %
Initial Query	17.61	-	-	1.15	-	-
Index Opt.	-	-	-	-	-	-
Query Opt.	10.87	38%	25%	0.39	66%	25%
Schema Opt.	8.41	22.6%	99.8%	0.33	15.4%	99.8%
Memory Opt.	7.89	6%	-	0.3	9%	-

Query 2	Before Cache			After Cache		
	Time	Time %	Space %	Time	Time %	Space %
Initial Query	1535	-	-	1463	-	-
Index Opt.	7.83	99.4%	-	0.62	99.9%	-
Query Opt.	1.71	78%	-	0.17	72.5%	-
Schema Opt.	1.38	19.2%	99.8%	0.15	11.8%	99.8%
Memory Opt.	1.29	6.5%	-	0.13	13.3%	-

Query 3	Before Cache			After Cache		
	Time	Time %	Space %	Time	Time %	Space %
Initial Query	0.36	-	-	0.07	-	-
Index Opt.	0.23	36%	-	0.01	85.7%	-
Query Opt.	0.19	17%	-	0.01	0%	-
Schema Opt.	0.14	26%	99.8%	0	100%	99.8%
Memory Opt.	0.12	14%	-	0	0%	-

Query 4	Before Cache			After Cache		
	Time	Time %	Space %	Time	Time %	Space %
Initial Query	10.41	-	-	0.27	-	-
Index Opt.	-	-	-	-	-	-
Query Opt.	-	-	-	-	-	-
Schema Opt.	6.96	33.1%	99.8%	0.16	40.7%	99.8%
Memory Opt.	6.57	5.6%	-	0.13	18.75%	-

Query 5	Before Cache			After Cache		
	Time	Time %	Space %	Time	Time %	Space %
Initial Query	14.96	-	-	0.44	-	-
Index Opt.	-	-	-	-	-	-
Query Opt.	4.82	67.7%	-	0.24	45%	-
Schema Opt.	3.37	30%	99.85%	0.2	16.67%	99.85%
Memory Opt.	2.34	30%	-	0.19	5%	-

3.2 Database Size Effect

The following plots show the effect of increasing database sizes on the execution time of our 5 queries. We consider both before and after *disk cache*.

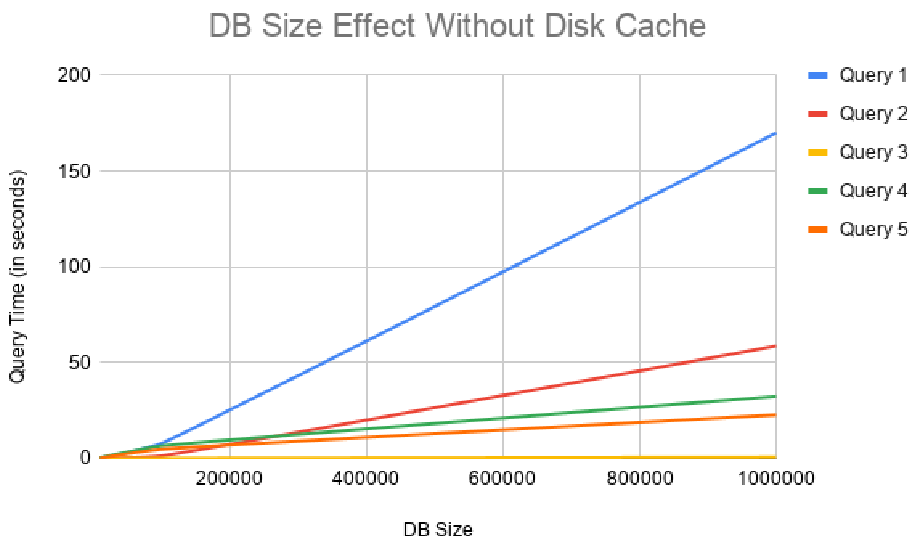

Figure 30: Database Size Effect Without OS (Disk) Cache.

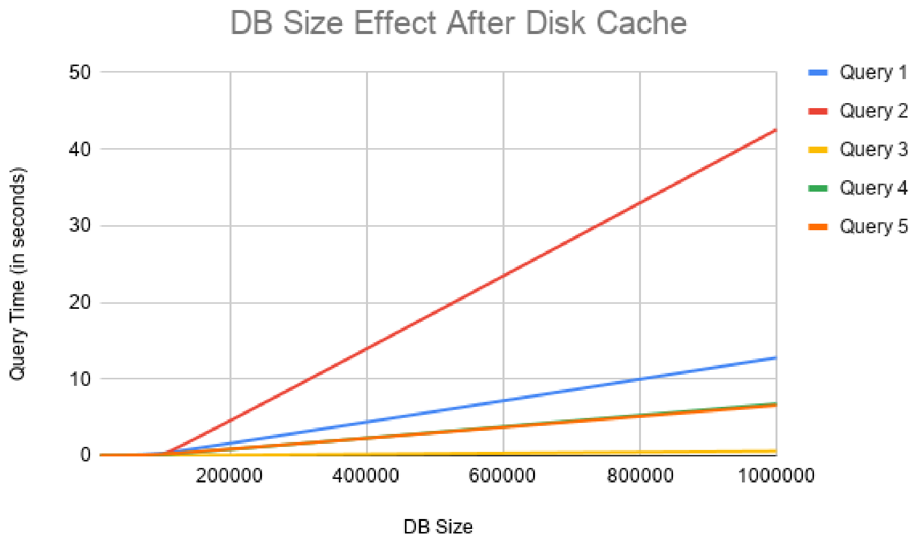

Figure 31: Database Size Effect After OS (Disk) Cache.

3.3 Optimized SQL vs. NoSQL

The following plot shows the different in execution time of each query between optimized *SQL* and *NoSQL*. The comparison is done on a database with $100,000$ records per table.

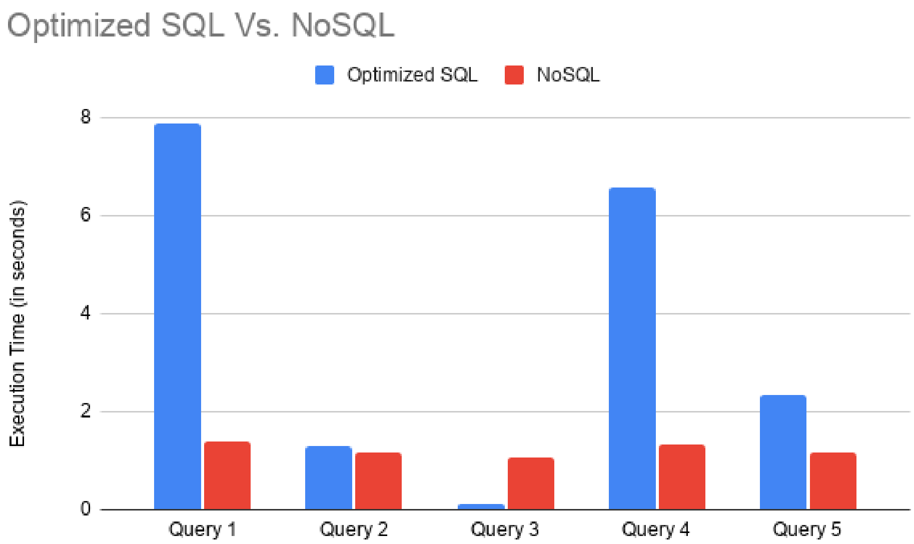

Figure 32: Comparison between optimized SQL and NoSQL on 100,000 records per table.

3.4 Hardware Effect

We conducted a hardware comparison between to devices of different specifications. The specifications are as follows :

- **Device (1) :**

  - **Operating System :** `Ubuntu 20.04`.
  - **CPU :** `Intel i5 6600k (4 cores)`.
  - **RAM :** `16GB`.
  - **Disk Type :** `HDD`.
- **Device (2) :**
  - **Operating System :** `Ubuntu 18.04`.
  - **CPU :** `Intel i7 4510u (2 cores)`.
  - **RAM :** `8GB`.
  - **Disk Type :** `HDD`.

Query Number	Before Cache		After Cache	
	Device (1)	Device (2)	Device (1)	Device (2)
Query 1	7.89	12.05	0.3	0.39
Query 2	1.29	29.16	0.13	0.81
Query 3	0.12	0.17	0	0.03
Query 4	6.57	9.27	0.13	0.2
Query 5	2.34	6.01	0.19	0.27

Time is measured in *seconds*. We can see that the **better** the hardware, the **faster** the query executes, both *with* and *without* disk cache.

4 Final Remarks

In this work, we have discussed various database optimization techniques and showed how the query execution time can be affected for both *SQL* and *NoSQL* databases. The final remarks can be summarized as follows :

- With good optimization, *SQL* databases can achieve a comparable performance with *NoSQL* databases.
- **MySQL** is a very versatile *DBMS* that can adapt to multiple optimization operations, enabling the user to increase queries execution speed.
- **MongoDB** can be the perfect choice for a *NoSQL DBMS*, as it's easy to use and deploy on very large systems.
- For some operations, *index optimization* can provide a huge performance improvement, if it's done right on specific fields in the database.
- It's a good practice to extract *semantic* and *statistical* heuristics based on your database. This can provide the developer with good insights on how to optimize queries and eliminate redundant operations.
- Try to keep the record size in frequently-accessed tables *fixed*, as it's much faster to access fixed-size records. Moreover, it's better to partition the infrequent variable-size fields into separate tables.