

Assignment 2: Benchmarking

Sashank Silwal

Special Topics in Computer Science: Big Data Systems

CS-UH 3260 Spring 2023

Database initialization and loading:

Initilization Times:

MonetDB: 3.779 sec

MYSQL: 4.290 sec

```
(base) → Assignment_2 time mysql -uroot -p -D flight --local-infile=1 < flights/air_ddl.sql

Enter password:
mysql -uroot -p -D flight --local-infile=1 < flights/air_ddl.sql 0.01s user 0.01s system 0% cpu 4.290 total
(base) → Assignment_2 time mclient -u monetdb -d flight < flights/air_ddl.sql

password:
operation successful
mclient -u monetdb -d flight < flights/air_ddl.sql 0.01s user 0.01s system 0% cpu 3.779 total
(base) → Assignment_2
```

Loading Times:

MonetDB: 54.295 sec

MYSQL: 3 min 33.85 sec

```
(base) nyuad@C2-LIB-M1-09 Assignment_2 % time mclient -u monetdb -d flight < flights/load_monetdb.sql

password:
12167426 affected rows
mclient -u monetdb -d flight < flights/load_monetdb.sql 0.01s user 0.01s system 0% cpu 54.295 total
(base) nyuad@C2-LIB-M1-09 Assignment_2 %
```

```
(base) nyuad@C2-LIB-M1-09 Assignment_2 % time mysql -uroot -p -D flight --local-infile=1 < flights/load_mysql.sql

Enter password:
mysql -uroot -p -D flight --local-infile=1 < flights/load_mysql.sql 0.96s user 14.86s system 7% cpu 3:33.85 total
(base) nyuad@C2-LIB-M1-09 Assignment_2 %
```

Table Status:

MonetDB:

```
sql>select * from ontime limit 1;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| year | quar | mont | dayo | dayo | flightdate | uniq | airlin | carri | tailnum | fligh | origin |>
: d   : ter  : hd   : fmon : fwee :           : ueca : eid   : er   :         : tnum  : airpor :>
:     :      :      : th   : k    :           : rrie :       :      :         :      : tid    :>
:     :      :      :      :      :           : r    :       :      :         :      :        :>
+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+
| 2021 | 1    | 1    | 3    | 7    | 2021-01-03 | 9E   | 20363 | 9E   | N607LR | 4628 | 11193 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 tuple !97 columns dropped!
note: to disable dropping columns and/or truncating fields use \w-1
sql>select count(*) as c from ontime;
+-----+
| c      |
+=====+
| 12167426 |
+-----+
1 tuple
sql>
```

MySQL

```
mysql> SHOW TABLE STATUS;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| Name | Engine | Version | Row_format | Rows | Avg_row_length | Data_length | Max_data_length | Index_length | Data_free | Auto_increment | Create_time | Update_time | Check_time |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| ontime | InnoDB | 10 | Dynamic | 11960320 | 256 | 3064987648 | 0 | 0 | 4194304 | NULL | 2023-03-04 16:19:48 | 2023-03-04 16:24:42 | NULL |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

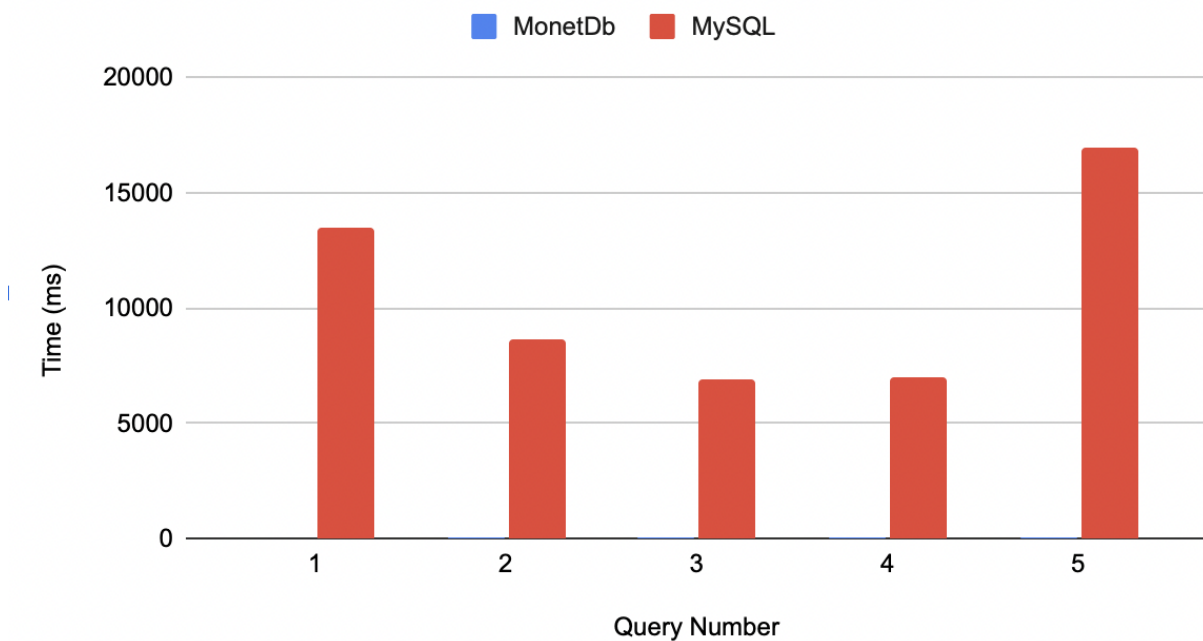
mysql>
```

Benchmarking:

Based on the histograms and the execution times of the five queries, it is evident that MonetDB performs better for every query in our dataset. MonetDB's columnar storage model allowed for a more efficient data aggregation operations for the queries.

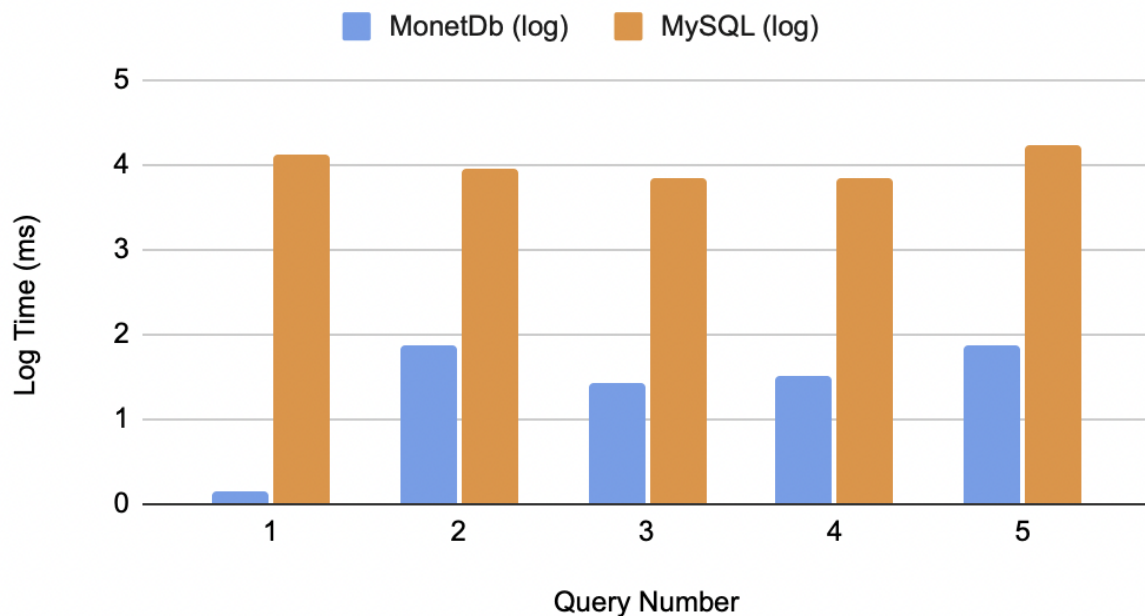
Query	MonetDb (ms)	MySQL (ms)
1	1.385	13490
2	72.252	8630
3	25.579	6888
4	31.759	6980
5	72.016	16980

Graph showing time for the queries in MonetDb and MySQL



Taking the log of time

Log time (ms) for MonetDb MySQL



Propose ways to improve the performance of the slower system on one of the queries. **(1point)**

To improve the performance of query 2:

```
-- Q2: Average monthly flights
SELECT avg(c1) FROM (
  SELECT YearD, MonthD, count(*) AS c1
  FROM ontime
  GROUP BY YearD, MonthD
) as tmp;
```

Adding indexes on the YearD and MonthD columns can improve the performance of the query by reducing the number of rows that need to be scanned during the GROUP BY and COUNT operations.

```
CREATE INDEX ix_ontime_YearD_MonthD ON ontime (YearD, MonthD);
```

After adding the indexes Q2. now completes in **2.60 sec**

```
mysql> SELECT avg(c1) FROM (
->     SELECT YearD, MonthD, count(*) AS c1
->     FROM ontime
->     GROUP BY YearD, MonthD
-> ) as tmp;
+-----+
| avg(c1) |
+-----+
| 529018.5217 |
+-----+
1 row in set (9.52 sec)

mysql> CREATE INDEX ix_ontime_YearD_MonthD ON ontime (YearD, MonthD);
Query OK, 0 rows affected (18.48 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql> SELECT avg(c1) FROM (     SELECT YearD, MonthD, count(*) AS c1     FROM ontime     GROUP BY YearD, MonthD ) as tmp;
+-----+
| avg(c1) |
+-----+
| 529018.5217 |
+-----+
1 row in set (2.60 sec)
```

Part III

To simulate the concurrency control issues, we will modify the isolation level of MySQL and perform transactions that cause these issues.

Phantom Read:

- Client 1 executes a SELECT query with READ COMMITTED isolation level, which means it should only see data that has been committed by other transactions.
- Client 2 inserts a new row into the same table that satisfies the WHERE clause of Client 1's SELECT query, after Client 1 has already executed its query.
- When Client 1 re-executes the same SELECT query, it sees the newly inserted row that was not present in the first execution, as it meets the WHERE clause criteria, creating the appearance of a phantom row that was not there before.

In Client 1:

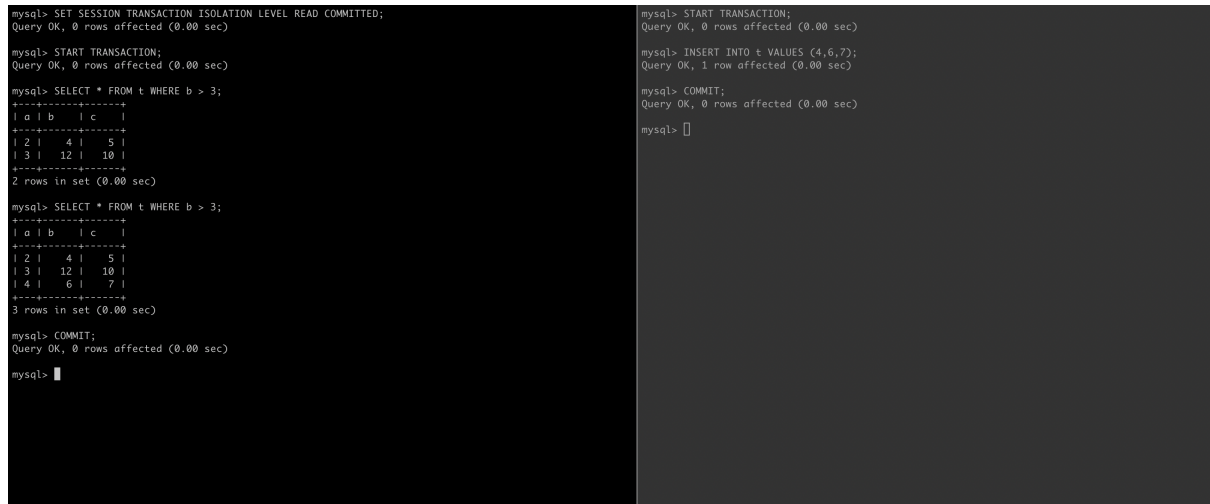
```
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
SELECT * FROM t WHERE b > 3;
```

In Client 2:

```
START TRANSACTION;
INSERT INTO t VALUES (4,6,7);
COMMIT;
```

In Client 1:

```
SELECT * FROM t WHERE b > 3;  
COMMIT;
```



The screenshot shows two MySQL terminal windows. The left window shows Client 1's queries: setting the isolation level to READ COMMITTED, starting a transaction, and executing a SELECT query twice. The first SELECT returns 2 rows, and the second returns 3 rows. The right window shows Client 2's queries: starting a transaction, inserting a new row (4,6,7), and committing. The SELECT query in Client 1 is executed before Client 2's insert, so it only sees the first two rows.

```
mysql> SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;  
Query OK, 0 rows affected (0.00 sec)  
  
mysql> START TRANSACTION;  
Query OK, 0 rows affected (0.00 sec)  
  
mysql> SELECT * FROM t WHERE b > 3;  
+-----+  
| a | b | c |  
+-----+  
| 2 | 4 | 5 |  
| 3 | 12 | 10 |  
+-----+  
2 rows in set (0.00 sec)  
  
mysql> SELECT * FROM t WHERE b > 3;  
+-----+  
| a | b | c |  
+-----+  
| 2 | 4 | 5 |  
| 3 | 12 | 10 |  
| 4 | 6 | 7 |  
+-----+  
3 rows in set (0.00 sec)  
  
mysql> COMMIT;  
Query OK, 0 rows affected (0.00 sec)  
  
mysql>  
  
mysql> START TRANSACTION;  
Query OK, 0 rows affected (0.00 sec)  
  
mysql> INSERT INTO t VALUES (4,6,7);  
Query OK, 1 row affected (0.00 sec)  
  
mysql> COMMIT;  
Query OK, 0 rows affected (0.00 sec)  
  
mysql>
```

Unrepeatable read

- In Client 1, a SELECT query is executed with READ COMMITTED isolation level, searching for a row with a value of 1 in the 'a' column of the table 't'.
- In Client 2, an UPDATE query is executed, incrementing the value of 'b' column by 1 for the same row that satisfies the WHERE clause of the previous query.
- Before Client 1 completes the first SELECT query, Client 2 commits its UPDATE transaction, changing the value of the row being read by Client 1.
- Client 1 then executes another SELECT query for the same row, but this time the result is different from the previous query because the row was updated by Client 2 after the first SELECT query was executed. Same query executed twice at different points in time returns different results.

In Client 1:

```
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;  
START TRANSACTION;  
SELECT * FROM t WHERE a = 1;
```

In Client 2:

```
START TRANSACTION;
UPDATE t SET b = b + 1 WHERE a = 1;
COMMIT;
```

In Client 1:

```
SELECT * FROM t WHERE a = 1;
COMMIT;
```

The screenshot shows two MySQL terminal windows side-by-side. The left window represents Client 1, and the right window represents Client 2.

Client 1 (Left Window):

```
mysql> SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
Query OK, 0 rows affected (0.00 sec)

mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM t WHERE a = 1;
+-----+
| a | b | c |
+-----+
| 1 | 4 | 3 |
+-----+
1 row in set (0.01 sec)

mysql> SELECT * FROM t WHERE a = 1;
+-----+
| a | b | c |
+-----+
| 1 | 5 | 3 |
+-----+
1 row in set (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)

mysql>
```

Client 2 (Right Window):

```
mysql>
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE t SET b = b + 1 WHERE a = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)

mysql>
```

The sequence of events is as follows: Client 1 starts a transaction and reads a row with a=1, b=4. Then Client 2 starts a transaction and updates the same row to b=5. Finally, Client 1 commits its transaction, reading the row again and seeing b=5, which is a dirty read because it reads uncommitted data from Client 2's transaction.

Dirty Read

- In Client 1, a SELECT query is executed with READ UNCOMMITTED isolation level, which allows it to read data that has not yet been committed by other transactions. It searches for a row with a value of 1 in the 'a' column of the table 't'.
- In Client 2, an UPDATE query is executed, setting the value of 'b' column to 6 for the same row that satisfies the WHERE clause of the previous query.
- Before Client 2 commits its UPDATE transaction, Client 1 executes another SELECT query for the same row, which returns the value that has not yet been committed by Client 2's transaction. This scenario is called a dirty read because Client 1 reads uncommitted data that may be rolled back later.
- Client 1 then commits its transaction, which may cause inconsistencies in the data because it is based on the uncommitted data read earlier.

In Client 1:

```
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
START TRANSACTION;
SELECT *FROM t WHERE a = 1;
```

In Client 2:

```
START TRANSACTION;
UPDATE t SET b = 6 WHERE a = 1;
```

In Client 1:

```
SELECT * FROM t WHERE a = 1;
COMMIT;
```

```
mysql> SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
Query OK, 0 rows affected (0.00 sec)

mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM t WHERE a = 1;
+-----+-----+
| a | b | c |
+-----+-----+
| 1 | 5 | 3 |
+-----+-----+
1 row in set (0.00 sec)

mysql> SELECT * FROM t WHERE a = 1;
+-----+-----+
| a | b | c |
+-----+-----+
| 1 | 6 | 3 |
+-----+-----+
1 row in set (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)

mysql>

mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE t SET b = 6 WHERE a = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0

mysql>
```

The SELECT statement in Client 1 will return the updated value of b even though the update has not yet been committed by Client 2.

Lost Update:

- In Client 1, an UPDATE query is executed, setting the value of 'b' column to 10 for the row where 'a' column equals 2.

- In Client 2, another UPDATE query is executed, setting the value of 'b' column to 20 for the same row that satisfies the WHERE clause of the previous query.
- Before either transaction completes, Client 1 commits its UPDATE transaction, overwriting the value set by Client 2 in the same row.
- Client 2 then commits its UPDATE transaction, but the value it set for the row is lost because Client 1's transaction had already committed with a different value.

In Client 1

```
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
SELECT b FROM t WHERE a = 2;
UPDATE t SET b = 10 WHERE a = 2;
```

In Client 2

```
START TRANSACTION;
SELECT b FROM t WHERE a = 2;
UPDATE t SET b = b + 10 WHERE a = 2;
```

In client 1

```
COMMIT;
```

In client 2

```
COMMIT;
```

In client 1

```
SELECT * FROM t WHERE a = 2;
```

```
mysql> SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
Query OK, 0 rows affected (0.00 sec)

mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE t SET b = 10 WHERE a = 2;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM t WHERE a = 2;
+-----+-----+
| a | b | c |
+-----+-----+
| 2 | 20 | 5 |
+-----+-----+
1 row in set (0.00 sec)

mysql>
```

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE t SET b = 20 WHERE a = 2;
Query OK, 1 row affected (8.57 sec)
Rows matched: 1 Changed: 1 Warnings: 0

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)

mysql>
```

- However, in case of MySQL, it uses a shared lock for read operations, which prevents other transactions from updating the same row until the shared lock is released.