CS223 Report: Part 1

1 Introduction

In this project I built a concurrent transaction simulator to mimic a real life transaction workload of an IoT based system on a database system. Two kinds of databases are used, *i.*, *e.*, PostgreSQL and MySQL, to compare their performances, and multi-threaded transactions are processed over multiple databases to mimic the concurrent process.

1.1 Data Model

In this project, data are divided into 3 categories – metadata, observation data and semantic observation data. Metadata is static which is related to users, space and sensors, thus should be inserted before observation data and semantic oberservation data. Observations is timeseries data generated from sensors. Totally, there are three types of sensors, *i*, *e.*, wifiap sensors, wemo sensors, and thermometer sensors, provided in the dataset. The timestamp attribute records the time that observations arriving at TIPPERS system. Based on these observation data, TIPPERS generates semantic observations, including presence and occupancy, which indicate the presence of people and the occupancy of buildings. The timestamp in semantic observations is associated with the arrival time of the semantic observations to the system.

1.2 Definition of Transaction

In this project, we define two kinds of transaction, the query transaction, and the insert transaction. Each query transaction contains one query statement, and an insert may contain multiple statements. That is because query processing does not change databases, and executing queries one by one will not affect atomicity and consistency. However, database will be updated after insertions, thus atomicity and consistency are required. Thus, we use different settings for different operations to mimic the actual situations. A set of inserts in a time period, such as, 60 minutes or 24 hours, is modeled as a transaction. That is, insertions from all sensors in a given time period belong to a same transaction. We can also define transactions based on timestamps and sensor id, however, a user activity may be captured by multiple sensors at the same time, thus should be considered as one transaction. Furthermore, semantic observations are generated based on observation data thus should be regarded as one transaction, but they share different sensor ids in the dataset. Considering of this we define transactions only based on timestamps in the statements [1].

2 Evaluations

We process concurrent transactions on PostgreSQL and MySQL to evaluate their performances. We mainly test the influence of the following parameters: the number of con-

[1] In project 2 transactions are defined based on (sensor-id, timestamp) to mimic distributed transactions.

current threads, isolation level, and data size. Performances are evaluated with throughput (*i, e.*, number of transactions per second), average response time for queries, and average response time for the total workload Without specific explanations, the default values of these parameters are set to 4, 'REPEATABLE READ', and 'low' [2], respectively.

2.1 Number of Concurrent Threads

In this experiment we set the number of concurrent thread to 2, 4, 6, 8, 10 and test the performances of the system. Results of PostgreSQL and MySQL are shown in Table 1 and Table 2, respectively. Throughput for insertions in PostgreSQL is first increased and then decreased with the number of threads increasing. Similarly, the average response time for queries and the overall average response time for the whole workload are firstly decreased then increased. Throughput for insertions in MySQL is first deceased and slightly increased with the number of threads increasing. The average response time for queries and the overall average response time for the whole workload are changed slightly, but with when the number of concurrent threads is increased to 10, the performance becomes obviously worse. In this experiment, PostgreSQL outperforms MySQL.

	2	4	6	8	10
Throughput	62.29	87.49	49.44	46.56	26.81
Avg response time for queries	0.0037	0.0027	0.0031	0.0038	0.0051
Overall response time	0.0065	0.0048	0.0066	0.0092	0.0098

Table 1: PostgreSQL Performances with varing # of concurrent threads

	2	4	6	8	10
Throughput	43.16	28.21	30.76	34.32	27.26
Avg response time for queries	0.0053	0.0046	0.0033	0.0049	0.0070
Overall response time	0.0094	0.0102	0.0104	0.0114	0.01402

Table 2: MySQL Performances with varing # of concurrent threads

2.2 Isolation Level

In database systems, the isolation level can be SERIALIZABLE, REPEATABLE READ, READ COMMITTED, and READ UNCOMMITTED, indicating complexities of the locking strategy. In this experiment we test the influences of the isolation level towards the performances of the system. Results of PostgreSQL and MySQL are shown in Table 3 and Table 4, respectively.

	SERIALIZABLE	REPEATABLE READ	READ COMMITTED	READ UNCOMMITTED
Throughput	88.34	70.20	59.42	68.12
Queries: res time	0.0018	0.0027	0.0019	0.0026
Overall res time	0.0040	0.0053	0.0053	0.0057

Table 3: PostgreSQL Performances with varing isolation level

	SERIALIZABLE	REPEATABLE READ	READ COMMITTED	READ UNCOMMITTED
Throughput	29.82	39.56	35.42	44.01
Queries: res time	0.0039	0.0042	0.0050	0.0030
Overall: res time	0.0094	0.0091	0.0112	0.0066

Table 4: MySQL Performances with varing isolation level

[2]Consistent with the name in the dataset file, which means a smaller dataset.

In the experiments above we run experiments in a seqneunce of SERIALIZABLE, REPEATABLE READ, READ COMMITTED, and READ UNCOMMITTED. However, we found that this running procedure will influence performances of databases due to high frequency read and write on a laptop, especially for PostgreSQL. So we run the experiments again in a reverse order, and record the results in Table 5 and Table 6.

	SERIALIZABLE	REPEATABLE READ	READ COMMITTED	READ UNCOMMITTED
Throughput	57.13	67.52	55.94	43.37
Queries: res time	0.0026	0.0031	0.0027	0.0059
Overall res time	0.0061	0.0061	0.0062	0.0117

Table 5: PostgreSQL Performances with varing isolation level

	SERIALIZABLE	REPEATABLE READ	READ COMMITTED	READ UNCOMMITTED
Throughput	29.01	39.42	39.31	34.54
Queries: res time	0.0028	0.0039	0.0041	0.0043
Overall: res time	0.0086	0.0089	0.0090	0.0106

Table 6: MySQL Performances with varing isolation level

After getting an average of these two experiments we suggest READ UNCOMMITTED (for PostgreSQL) or SERIALIZABLE (for MySQL) for frequent insertions, and SERIALIZABLE for queries.

2.3 Data Size

In this experiment we test the influence of the data size. Consistent with the names in the data file, here we use "high concurrency" and "low concurrency" for these two dataset. In this experiment, the data size of high concurrency is about double the data size of low concurrency. Results for PostgreSQL and MySQL are shown in Table 5 and Table 6. Experiments with higher concurrency show higher throughput as well as longer average response time. And the query time is influenced much by the data size due to more data having been inserted in the high concurrency experiments. Overall, PostgreSQL outperforms MySQL.

	low	high
Throughput	66.78	84.94
Avg response time for queries	0.0043	0.0178
Overall response time	0.0071	0.0159

Table 5: PostgreSQL Performances with varing level of concurrency

	low	high
Throughput	42.54	66.10
Avg response time for queries	0.0042	0.2926
Overall response time	0.0092	0.18322

Table 6: MySQL Performances with varing level of concurrency