Comparative study between SQL and NoSQL solutions for smart grid applications

Emran Altamimi *Qatar University,*

***Abstract*—As the smart grid integrates more devices with higher sampling rates the amount of data generated becomes astronomical. The huge amount of data requires an efficient scale-able storage solution to take advantage of the data to provide timely insights and services to both consumers and electricity providers. In this work, a comparative study between SQL and No-SQL solutions is conducted on a comprehensive query types that are required for both consumers and utility providers. The scalability of both solutions on terms of both the amount of data and the workload is tested. Our results found that No-SQL solutions provided a clear superiority in all tests.**

***Index Terms*—Smart grid, MongoDB, MySQL, DBMS**

I. Introduction

THE Smart grid is filled with smart meters, electrical sensors and digital assets that results in a huge amount of data [1]. Such huge data made the power system more complex to operate especially with the integration of distributed energy sources, electric vehicles, and dynamic tarrif design [2]. Transforming this huge influx of data into insights and services for the grid utility requires advanced data analytics algorithms.

Roughly speaking, there are three main categories for data sources: customer data, grid data, and external data. The huge amount of data is thus characteristic of big data with wide varieties, frequencies, veracity, and different values range [3].

Customer data typically refers to information pertaining to customers such as electricity consumption and market data, while grid data refers to the data that is required to ensure the safe operation of the grid such as electrical measurements data. External data refers to any auxiliary data that helps with data analytics such as weather data.

In smart grids, older data management systems are often built on relational database systems, such as MySQL and Oracle [4]. As a consequence, many of these systems have begun to exhibit significant issues in dealing with the data flood generated by the smart grid while meeting the scalability needs of these systems. Furthermore, they are unable to manage the vast array of data sources in the smart grid, as well as the storage of unstructured and semi-structured data [5].

In the past years IT megacorporations such as SAP, EMC, Teradata, Oracle, SAS and IBM and grid giants like Toshiba LandisGyr, Electric\Telvent,Schneider,Siemens\eMeter, General Electric and ABB\Ventyx are providing products and services linked to electricity big data and smart energy management for both utilities as well as individuals. Furthermore, numerous firms that concentrate on energy management services that are data driven have seen fast development in recent years. The fast expansion of such businesses also highlights present energy system inefficiencies and people's rising desire for smart energy management, in addition to the potential role that big data analytics may play in supporting smart energy management.

Therefore the key expected benefits of NoSQL solutions on the context of the smart grid can be summarized in the following four points:

- Seamless integration of unstructured data such as questionnaires in the database. Questionnaires gives utilities crucial insights for their customers, in particular to better understand the responsiveness of customers to demand response programs. A seamless integration of such unstructured data in the database makes it easier for utilities to maintain and analyze the data.
- Better scalability. NoSQL provides an advantage over relational databases with their superior scalability capability. Electrical utilities would benefit immensely from better scalability as the number of consumers increases. Another source of big data that requires a highly scale-able system is the integration of huge amount of high resolution sensors.
- Sharding for high availability of the data. This becomes highly relevant for sensor data that gives critical information about the safe operation of the grid. Therefore, only relevant data is stored in their respective electrical stations to ensure high availability of the data.
- Lays the infrastructure of smart city services. Smart city services refers to digital services that effectively engages the citizens with digital services. For example, an electrical utility may provide insights for customers on how to save energy by analysing their consumption and habits.

To the best of our knowledge the research mostly focuses on forecasting and real time monitoring, energy efficiency optimization, demand side management, targeted marketing, energy consumption reports and consumption, and customer engagement.

II. Related work

The work on big data in the smart grid is still in its infancy, in particular, the efficacy of NoSQL databases for big data analytics in the smart grid is still largely unexplored. While big data technologies including NoSQL databases has been reviewed in the context of the smart grid, the works that provide a review of data management solutions are fairly limited. For example, the work in provides [6] a review of big data management in the context of the smart grid mentioning several real world platforms such as the implementation of Hadoop in Tennessee's smart grid. The authors in [3] discussed big data analytics management in several aspects. The authors highlighted numerous industrial solutions and products along side their intended usage. However, the review focused on the intended use cases of such products and did not review the high or low level infrastructure of the data management solutions. The authors in [7] provided a review of bi g data analysis in the aspects of data sources, analysis techniques and applications. The authors did not provide any information about the database systems. This issue extends to studies developing big data platforms for data analytics or for smart cities services. Some examples include the work in [8], [9], [10], however, only the high level presentation is available. These issues make it harder for future designers, especially within our scope of choosing the optimal database system in the context of the smart grid. In this section, we will review data management solutions in the context of the smart grid and review big data analytics solutions for the smart grid based on NoSQL databases.

Several aspects of the studies are highlighted; the choice of indexes and its rational, the use cases and queries in focus, a qualitative or quantitative comparison between NoSQL and SQL solutions, and migration guidelines if available.

The authors in [4] proposed a scheme to process electricity data based on MongoDB. The choice of indexes was not explicitly mentioned nor rationalized. The database was evaluated against MySQL and Hadoop Distributed File System (HDFS) on three different tests. The first test was the storage efficiency where MonogoDB was superior and demonstrated better scalability. For the second test the authors compared the databases on search performance with three increasing complexity SQL statements. No further information on the queries or meaning of complexity was mentioned. MongoDB performed better than MySQL however it slightly performed poorer than HDFS. For the final test the authors observed how the system handles faults, however no clear comparison was provided nor a quantitative evaluation.

The work in [11] proposed a framework where heterogeneous data is collected, pre-processed, and loaded to a NoSQL database where machine learning models for energy forecasting are trained. The NoSQL database used in their work was MongoDB, however, they did not evaluate the database performance in anyway.

The authors in [12] provided the lower level details of the architecture and design of a real world implementation of a smart city platform. The authors narrowed down the choice of NoSQL solution to document type databases because the data received from sensors was in JSON format and because they provide better efficiency and flexibility. The database system used was CouchDB which is a document oriented NoSQL solution less popular than MongoDB. All the data collected from sensors was saved as JSON document. However, a key distinction of CouchDB is that it has an HTTP REST API allowing for a more flexible interface which is appreciated in a smart city big data platform. The authors evaluated on two types of queries simple and complex. Simple queries are those queries which fetch the data of a single sensor, while complex queries fetch the data several or all sensors of a system. No further details about the index were mentioned

The work in [13] proposed a framework to extract features and data mine electricity consumption data and questionnaires. The selected database was MongoDB. No evaluations were done regarding the efficacy of the data storage solution.

The work in [14] focused on the feasibility of NoSQL solutions to provide for smart city services. While the authors provided a review for previous works of NoSQL benchmarks, they were not specific to the smart grid context or specific to time series data. For this reason we believe that while the authors concluded that Cassandra and HBase are the better databases overall, most literature in the smart grid opt for document based databases namely MongoDB.

The authors in [15] evaluated the performance of five databases: PostgreSQL, MySQL, MongoDB, Cassandra, and MSSQL on four different scenarios. The scenarios are writing and reading of either single or multiple wind turbine data. MongoDB performed best in all cases with Cassandra performing closest to it.

The work in [16] compared different storage and processing architectures for consumption data. The authors concluded that architectures built upon relational database management systems under-perform compared to distributed architectures. However, no comparison of the database systems was provided.

The authors in [17] evaluated the scalability performance of different data management solutions which include a data processing system (Hadoop and spark) and a database system (postgres-XL, HDFS, and cassandra. The authors first compared the loading time of the three databases and cassandra performed best and scaled better. The authors performed their analysis based on 7 query, however, they can be grouped on three main query types (aggregation, selection and filtering, and billing). Cassandra performed and scaled (increasing the number of sensors) better than HDFS on the aggregation query types. On billing and selection and filtering query types cassandra performed exceptionally better than HDFS. The authors also studied the effect of increasing the number of nodes on the same query types. Cassandra was superior on billing queries in the order of hundreds, even though it performed poorly compared to HDFS on the other queries. Our contribution lies in testing the performance of NoSQL and SQL solutions in the context of the smart grid. The performance is tested with regards to data scalability and workload scalability. We also provide details about the low level implementations of both MongoDB and MySQL and test them on 6 different comprehensive query types.

III. Dataset overview

The dataset used in this work revolves around a simulation of electricity consumption in Qatar. The dataset contains consumer-related data along with their electricity consumption readings at different timestamps. In addition, the dataset also contains data related to different zones in Qatar along with some temperature readings throughout the years which

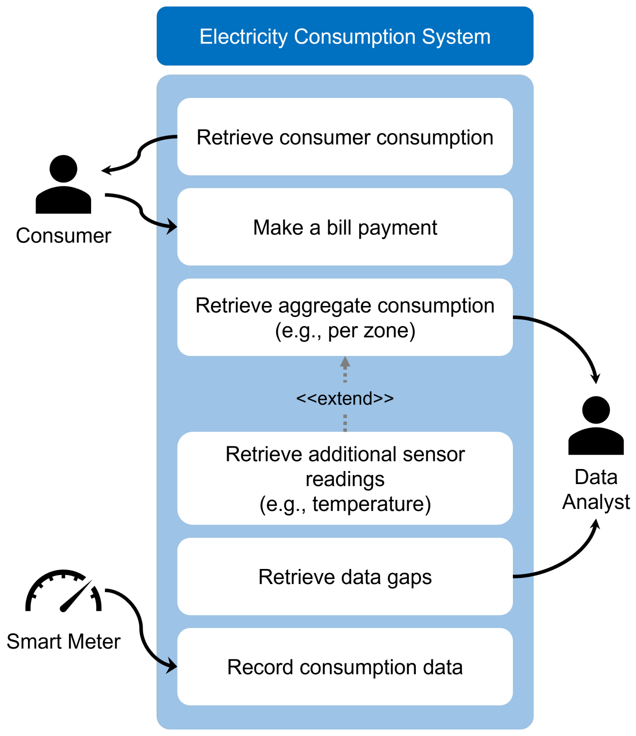
Electricity Consumption System
Retrieve consumer consumption
Make a bill payment
Retrieve aggregate consumption (e.g., per zone)
<<extend>>
Retrieve additional sensor readings (e.g., temperature)
Retrieve data gaps
Record consumption data
Consumer
Data Analyst
Smart Meter

Fig. 1. Use case diagram

can be used in the analysis. Moreover, the dataset contains information about billing rates and billing records for each consumer. The number of records for each entity is shown in table I.

TABLE I
NUMBER OF RECORDS FOR EACH ENTITY

Entity	Number of records
Consumers	500 to 4000
Consumption records	50000 to 400000
Temperature records	250 to 2000
Zones	4

IV. USE CASES

In our work and based on the smart meter data analytics research [18] we have identified 6 query types that correspond to all possible use cases and smart meter data analytics. This section will demonstrate the query types and their respective use cases. The use cases are summarized in Figure 1 and described in detail in table II. The SQL queries are shown in table II and NoSQL (MongoDB) queries are shown in Appendix A. The theoretical plan and cost analysis of the queries is presented in figure 6 in Appendix B.

A. Retrieve readings based on a date range and ID

In this use case, a date or time range is specified and the data for a specific set of consumers is returned. The first type of queries with the most abundant number of use cases is to retrieve data from a date range. Namely, utilities can build advanced forecasting models on past data to predict future consumption [19]. Utilities can also utilise advanced data mining techniques to uncover hidden patterns, for example, what are the peak hours in weekdays and how much does the change in pricing affected them [20].

B. Billing and bill payment

Based on the first query type there are the two most important use cases which are related to billing. The first is billing, in this case the query type is fetching the data from specific times then aggregating them and multiplying them with their respective tariff rates and the amount due is calculated. The second case is bill payment, where the consumer pays their due.

C. Retrieve readings on a zone basis

This query is important for several reasons, mainly to understand specific characteristics of customers within a certain geographic area. Another important concept is that numerous zones can be defined on many different basis, for example, on a substation basis, which would allow for better fault identification and localisation [21]. Furthermore, this same query is similar to many types of attributes that do not change over time, for example, is the consumer using renewable energy and what are the specifications.

D. Auxiliary data

One of the main appeal of advanced database systems for the smart grid is their ability to make use of other auxiliary data to improve the stability and reliability of the grid. For example, temperature data is an important auxiliary data that is often used in improving the performance of load forecasting [22]. Auxiliary data can also be unstructured data such as questionnaires which can only be stored on NoSQL databases [13]. We consider a query to fetch the auxiliary data in conjunction with other data.

E. Missing and anomalous data points

The large amount of sensors and huge data makes it almost inevitable that there will be missing and anomalous data. Retrieving these data points is essential for improving the performance of the trained model and for correct analysis of the data. Furthermore, detecting these data points is essential to compact energy theft. In this query types, we fetch missing data points.

F. Inserting records

In this query type, acquired data is inserted.

TABLE II
QUERY TYPES AND USE CASES DESCRIPTION

Use case	Brief specification	Query example
Retrieve consumer monthly consumption	'Customer' can interact with the system to retrieve their electricity consumption for any specific month of their choice.	SELECT BILLING.ID, BILLING.CONSUMER_ID, BILLING.MONTH, BILLING.YEAR, BILLING.TOTAL, BILLING.AMOUNT_DUE, SUM(CONSUMPTION.CONSUMPTION) FROM BILLING, CONSUMPTION WHERE CONSUMPTION.CONSUMER_ID = 195 AND BILLING.MONTH = 'January' AND BILLING.YEAR = 2021 AND CONSUMPTION.TIMESTAMP > '2021-01-01 00:00:00' AND CONSUMPTION.TIMESTAMP < '2021-02-01 00:00:00';
Make a bill payment	Customer can register a payment and have their pending balance reduced with the amount that was paid.	UPDATE BILLING SET AMOUNT_DUE=0, PAID = TRUE WHERE CONSUMER_ID = 1 AND MONTH = 'June' AND YEAR = 2008;
Retrieve Consumption per zone	'Data analyst' can view electricity consumption for each zone in the country.	SELECT ZONE.ZONE_NAME, SUM(CONSUMPTION.CONSUMPTION) AS SUM_CONSUMPTION FROM ZONE INNER JOIN CONSUMER ON CONSUMER.ZONE_ID=ZONE.ZONE_ID INNER JOIN CONSUMPTION ON CONSUMER.ID = CONSUMPTION.CONSUMER_ID WHERE CONSUMPTION.TIMESTAMP > '2020-01-01 15:29:31' AND CONSUMPTION.TIMESTAMP < '2022-01-01 15:29:31' GROUP BY ZONE.ZONE_NAME;
Retrieve additional sensor readings	For AI based analysis, 'Data analyst' could retrieve additional sensor readings in conjunction with the consumptions grouped by each zone in the country.	select temp.zone_name, temp.avgtemp, cons.sumconsumption from (select zone.zone_name, avg(zonetemp.temperature) as avgtemp from zonetemp inner join zone on zonetemp.zone_id=zone.zone_id where zonetemp.timestamp > '2020-01-01 15:29:31' and zonetemp.timestamp < '2022-01-01 15:29:31' group by zone.zone_name) as temp inner join (select zone.zone_name, sum(consumption.consumption) as sumconsumption from zone inner join consumer on consumer.zone_id=zone.zone_id inner join consumption on consumer.id = consumption.consumer_id where consumption.timestamp > '2020-01-01 15:29:31' and consumption.timestamp < '2022-01-01 15:29:31' group by zone.zone_name) as cons on temp.zone_name = cons.zone_name;
Retrieving data gaps	To retrieve the timestamps of two consecutive data readings where the time between the two recordings exceed a threshold which is considered a gap in data reading. This could be used by 'Data analyst' to insert interpolated data into the database.	SELECT * FROM consumption WHERE consumption=0;
Record consumption data	'Smart Consumption Meter' installed in customer homes can push consumption readings into the system.	INSERT INTO CONSUMPTION (id, timestamp, consumer_id, consumption) VALUES (3, '2003-07-09 17:32:44', 434, 52.157731947954744);

V. CAP THEOREM

When dealing with data on a national scale with millions of users, the issue of scalability and performance becomes critical. Especially that NoSQL databases are limited by the CAP theorem which states that only two of consistency, availability or partition tolerance can be guaranteed by an NoSQL database. The following are the three CAP scenarios and their implications when dealing with power consumption data.

1) CA: consistency and availability: In this case, the database can ensure that the energy readings for both the consumer and the utility are consistent, which is critical for billing purposes. Another concern is availability, which is less important than consistency for billing purposes, but ensuring high availability on a national scale is important. Because sub-optimal availability would affect tens of thousands of customers and results on perceived lower quality of service. However, in this case, the database is not network partition tolerant, which is arguably more significant because utilities must ensure that their service is not interrupted by network or node failures.
2) CP: consistency and network partition tolerant: Sacrificing availability might be problematic because the scale of application for electrical utility is huge and will affect a large portion of the population.
3) AP: Availability and network partition tolerance: Arguably the worst choice, because consistency is the most important issue in any financial transactions. The system's high availability would also be rendered meaningless because consumers would spend the majority of their time disputing false billing transactions.

As a result, the most important attributes for database engineers in electrical utilities are consistency and availability or consistency and partition tolerance. This is why for most of the applications discussed in the **??** section, MongoDB was the preferable choice which prioritizes consistency and partition tolerance. MySQL was another popular choice, which depending on the configuration either prioritizes consistency and availability or consistency and partition tolerance. By default, however, MySQL prioritises consistency and availability because it follows a master-slave paradigm.

VI. Relational database model

A. SQL database overview

A SQL database was constructed for the electricity consumption dataset using MySQL platform. The database consists of 6 tables which are:

1) Zone: contains basic information about different zones in Qatar.
2) ZoneTemp: contains temperature readings in each zone at different timestamps.
3) Consumer: contains information about electricity consumers.
4) Consumption: contains consumption data for each consumer at different timestamps.
5) Billing_rates: contains the billing rates for each of the weekdays.
6) Billing: contains the monthly billing information for each consumer.

Figure 2 shows the entity relations (ER) diagram and Figure 3 shows the schema for the electricity consumption database.

B. SQL database indexing

The default index used in MySQL databases is created on the primary key of each table. The type of default index is B-Tree. Due to the fact that the fundamental data in the electricity database is the electricity consumption records with time, which is considered time-series data, most of the insertion and retrieval queries will be related to certain time intervals. Therefore, the most appropriate attribute to index is the timestamp in the consumption table. Furthermore, when retrieving consumption data, time-related and range queries will be the most common queries. Thus, the most suitable indexing type is B-Tree which works best for range queries.

VII. NoSQL model (MongoDB)

This section will discuss how the architecture of MongoDB affects the decision makers and database designers. Several aspects will be discussed in this section which are replication and fault tolerance, sharding, time-series considerations and recommendations, and concurrent writes and reads.

A. Replication and fault tolerance

MongoDB follows the master-slave paradigm with automatic fail-over to provide redundancy and availability. Every replica set has only one master (primary) node on which all writes operations are pushed to. Slave (secondary) nodes then replicate the primary node and in case of a failure in the primary node, secondary nodes hold an election to vote for a new primary node. Figure **??** shows the two In the case of a primary node failure the replica set is unavailable for write operations until consensus is reached (which typically has a median of 12 seconds). This, however, is of little concern since write operations for smart meters are in the order of 10 minutes to 1 hours. Read operations can still be performed on secondary nodes to ensure higher availability for data analytics purposes, in which real time processing is usually of little concern.

B. Sharding

Sharding is an effective technique to horizontally scale the database through dividing the same dataset across several nodes. The subset of a dataset is called a shard which is typically stored in a replica set. A query router then provides the necessary interface between the sharded cluster and the application. In MongoDB the shards are determined based on a shard key which is a field/s in a document. Sharding in the context of the electricity utility providers facilities an easier way to manage the scalability, efficiency, and performance of the sharded cluster. A sharding strategy is to shard the data based on location, allowing for faster performance since all models are trained on similar consumer anyway [23].

C. Concurrency

Concurrency is handled in MongDB by locking the documents when a write happens. Hierarchical locking is also possible in MongoDB, for example, a collection or the whole database can be locked.

D. NoSQL database overview

A NoSQL database was constructed for the electricity consumption dataset using JavaScript to convert the dataset to MongoDB compatible JavaScript Object Notation (JSON) format with all numeric primary keys replaced with MongoDB's Object Identifier. The database consists of 6 collections similar to the 6 tables depicted in Figure 2 which are:

1) Zone: contains basic information about different zones in Qatar.
2) ZoneTemp: contains temperature readings in each zone at different timestamps.
3) Consumer: contains information about electricity consumers.
4) Consumption: contains consumption data for each consumer at different timestamps.
5) Billing rates: contains the billing rates for each of the weekdays.
6) Billing: contains the monthly billing information for each consumer.

E. NoSQL database indexing

MongoDB uses B-tree datastructure to store indexes similar to the case of SQL database [24]. By default, each document has an indexed _id field that uniquely identifies a document in a collection. MongoDB also provides capabilities to add custom indexes of single fields or compound fields for boosting performance based on queries. The timestamp field was set similar to MySQL in the electricity consumption and zone temperature collections to improve performance for our selected use cases.

VIII. Comparison of the two solutions

After discussing both the SQL and NoSQL solutions key differences and advantages for NoSQL solutions are drawn. Mainly, NoSQL are attractive due to their inherit flexibility

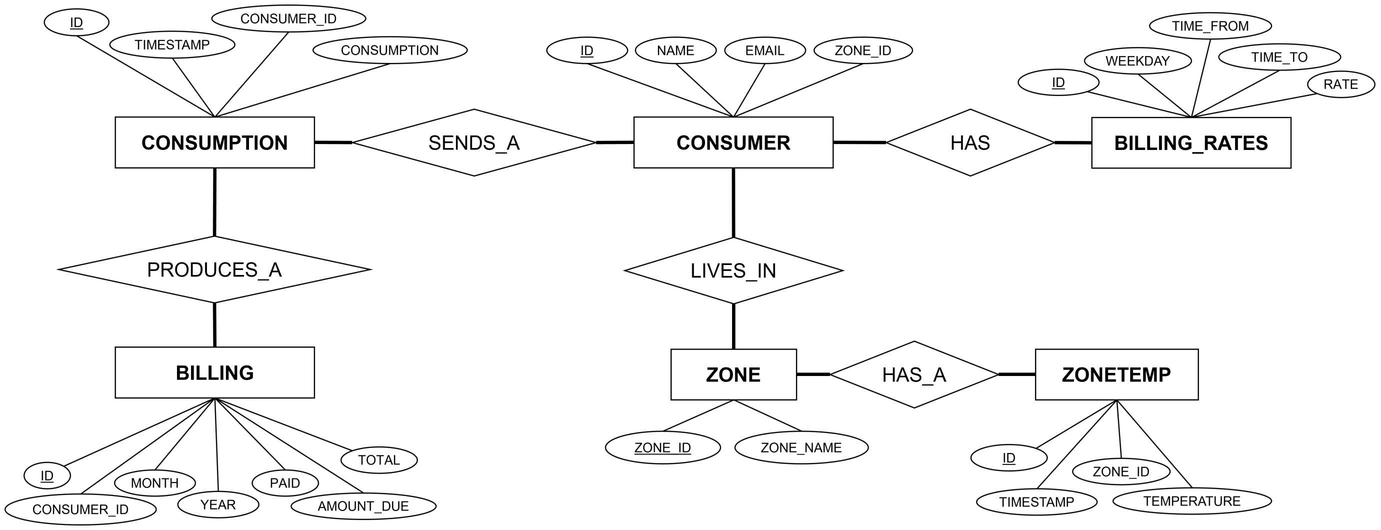

Fig. 2. ER diagram of the SQL database

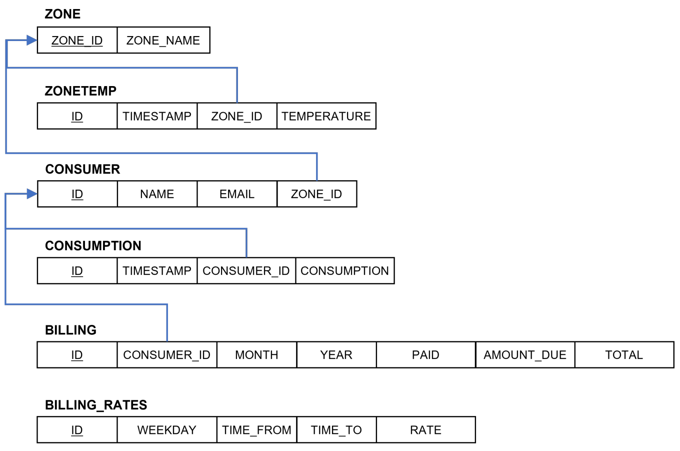

Fig. 3. Schema of the SQL database

of the data schema (for example unstructured data such as questionnaires). Such data schemas can be easily updated using NoSQL and new changes can be added such as adding a column to represent renewable energy sources . When it comes to making changes to the schema or performing updates, NoSQL grants users more flexibility than traditional relational databases do. To illustrate, the NoSQL solution has the potential to have a new column added to it with very little additional work required in order to store information about the generation of renewable energy. Because only a small percentage of people actually make use of renewable energy, it is of the utmost significance to have effective storage in NoSQL for the numerous attributes that might not be available for most consumers. SQL requires complex joins, which can slow down the execution of complex queries. The NoSQL solution, on the other hand, does not require joins making it faster. As an example, when trying to add new auxiliary data such as humidity for a more complex AI analysis, SQL requires joining the new tables with the old ones, which takes a lot of time and makes the process more complicated.

IX. Evaluation

A. Methodology

The purpose of this section is to compare and analyze the performance of the aforementioned systems for the processing of meter data query requests, the likes of which may be made in a smart grid. In order to accomplish this, we begin by generating data up to 400 meters in length with each measuring device comprising up to 100,0. After that, we will carry out two separate experiments. The first set will involve deploying the databases that were mentioned previously and expanding the size of the dataset from a total of 50,000 records to 400,000 measurements in order to determine how long it takes for each query to receive a response. In the second set of experiments, we increase the number of concurrent queries from five to six hundred and then evaluate each system's ability to handle an increasing workload with its various queries. *Experimental setup:* The dataset was constructed in JavaScript to JSON format. The queries and connection to both MySQL and MongoDB were simulated using Jmeter. Jmeter is a tool based completely on java used for measuring and analyzing the load on various services and applications. The experiments were done on ASUS TUF Gaming F15 with the specifications described in Table III.

TABLE III
DEVICE SPECIFICATIONS

Part	specification
CPU	Intel(R) Core(TM) i5-11300H @ 3.10GHz
RAM	24GB @ 3200MHz
GPU	RTX 3060
OS	windows 11

Queries: The queries discussed in details in section are used to evaluate both systems. The queries are meant to be comprehensive for all possible types of queries that might be possible in the context of the smart grid. *Data insertion:*

While setting both the databases, there was a noticeable difference in the speed and ease of inserting the data in MongoDB compared to MySQL. MongoDB was setup and ready in less than five minutes, while MySQL took around 30 minutes to setup.

B. Workload scalability

In the first experiment the 6 queries are tested on 5, 50, 100, 200, 400 and 600 concurrent connections. The connections were tested on 1000 consumers with 100,000 records. The average response data is reported for each query. Figure 4 shows a clear superiority for MongoDB both in terms of performance and scalability. MongoDB performs better for all workload simulations and scales linearly with a small slope compared to a larger slope for MySQL. However, in Query 1, the response rate actually increases exponentially with the workload. We attribute this reason to the complexity of the query, regarding how the storage schema is set up. Arguably, query 1 is the most expected query to have a high number of requests because of its very nature, that is, it is expected that the most common query types for consumers would be to retrieve their consumption given a date range. Therefore, an exponential growth for the response time for this query is unacceptable.

Moreover, we noticed during testing a high number of error rates for MySQL, and the experiments were re-run several times to ensure zero error rates. However, such issue was not tackled for MongoDB.

On the other hand, testing MySQL was easier and well supported by the testing community and debugging was not an issue. While on MongoDB numerous bugs were faced that required intensive knowledge about the testing tool and java language to be able to debug. For example, an issue we faced was that the connections to MongoDB were not automatically closed after the queries were satisfied. This led to a high error rate and an automatic closing of the sockets by MongoDB. Such issue was resolved by forcing the connections to be closed, which explicitly was mentioned in MongoDB API documentations to be used as a last resort.

Regarding the use cases, it is noticeable that the first query took the longest time when the workload was maximum in MySQL at an average of 7.8 seconds. On the other hand, the longest query type was the second query for MongoDB at 48 Millie seconds. However, when the workload was minimal the performance of both was fairly identical for both solutions, where MySQL even performed better in the second Query.

C. Database scalability

The second experiment tested the same 6 queries on both database solutions. The databases were scaled as mentioned in Table I. The workload was fixed on 400 concurrent connections. The average response rate is reported as can be seen in figure 5. The tests show again a clear show a clear superiority for MongoDB both in terms of performance and scalability. MongoDB performs better for all database simulations and scales linearly with a small slope compared to an exponential growth for MySQL for queries 1 and 4. However, other queries, the response rate increases linearly with scaling the database.

In this test, we noticed an increase error rates for MongoDB in the final database, that was only solved after addressing several other bugs. The issue of the lack of support from the testing community for NoSQL solutions hindered the progress, which slows progress and might be a deciding factors for many cloud solutions to opt for MySQL solutions.

Regarding the use-cases, it was noticeable that the starting point for MySQL was higher in order of magnitudes for query 1, 2, and 6 and fairly similar in the other queries. The first query had performed the worst at 14.3 seconds compared to the rest for MySQL, while query two was still the worst for MongoDB at 40 Millie seconds.

X. Guidelines on converting SQL tables and tuples to NoSQL collection

Converting the relational tables and data in the SQL database, we can first analyze the existing tables and start with the table that has no foreign references to convert. We then move to create other collections from tables that only refer to our first converted collection. We follow these steps until all tables have been converted to document-based collections.

As shown in Figure 3, we have six relational tables in total. The most basic table with only primary keys are the tables ZONE and BILLING_RATES. We first iterate through each of the items and load them as an array in a NodeJS application. The IDs generated for SQL are autoincrementing integers. However, MongoDB makes use of a hexadecimal ObjectID for primary keys. Using the MongoDB library for NodeJS, we can generate unique primary keys for each object and save the output to a JSON (Javascript Serializable Object Notation) file. A hashmap is used to store the mapping of SQL ID and ObjectID notations. The second stage is to similarly create collections for SQL tables that have only 'ZONE'.ID as a foreign key. Similar to the first step, we create JSON files for the tables 'CONSUMER' and 'ZONETEMP'. For the foreign key reference to zone_id, while we iterate through each item, we use the hashmap created for the zone table to retrieve the corresponding ObjectIDs and replace the SQL keys in the foreign key field. The same technique is applied to the remaining tables and foreign keys.

With the JSON files all prepared, MongoDB has an import utility that allows for importing JSON arrays into the database as a collection. A collection can be imported using the following command with the mongoimport utility:

```
mongoimport --uri "<database-url>
--type json
--file "<inputFile>.json"
{jsonArray
```

XI. Conclusion and future work

In this work, the performance of NoSQL and SQL solutions are tested in the context of the smart grid. The performance is

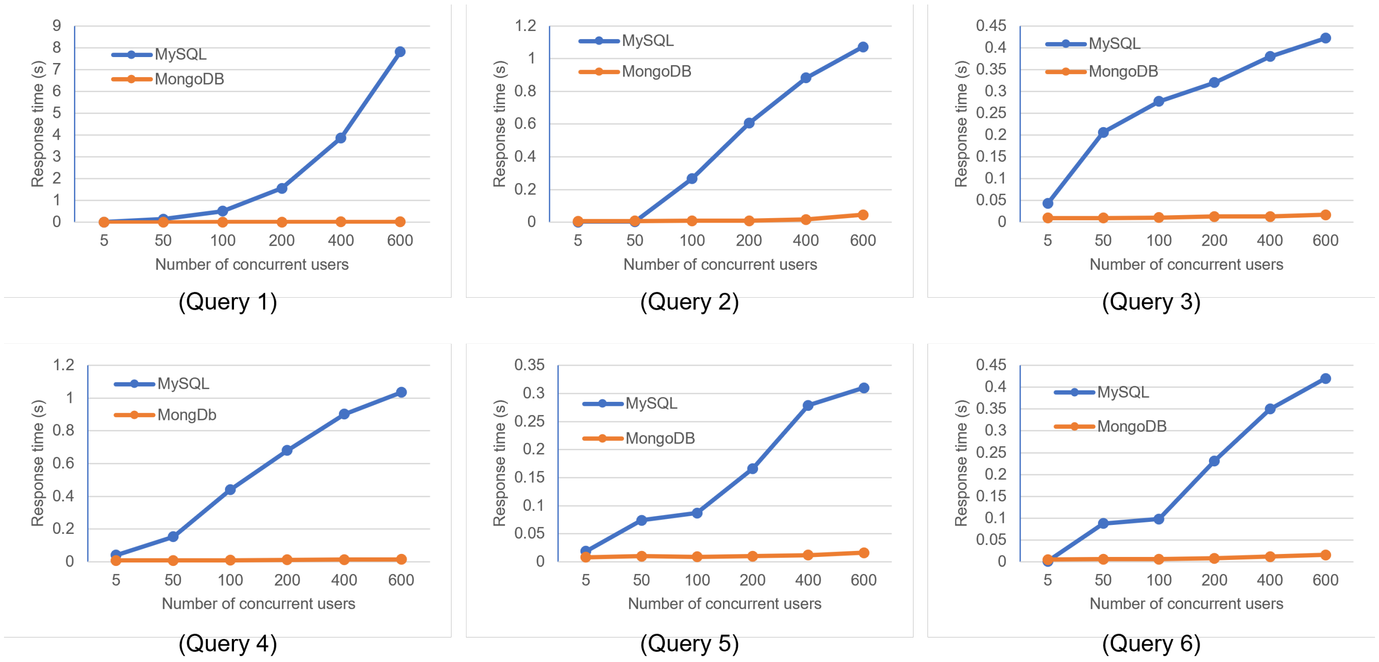

Fig. 4. Comparison between MySQL and MongoDB in workload scalability

tested with regards to data scalability and workload scalability. The tests were conducted on JMeter using six queries which are meant to be comprehensive and encompass all possible use-cases that might be possible in the context of the smart grid. The NoSQL solution was implemented in MongoDB and the SQL solution in MySQL. Our results indicate a clear superiority for NoSQL solution over SQL solution in terms of the response time. Furthermore, as the size of the data doubled, the response time for the NoSQL solution was fairly linear with a lower slope. On the other hand, the SQL solution suffered from the scalability issues as the response time increased with a higher slope and even in an exponential manner in some queries. Regarding the workload testing, similar behaviour is exhibited for both solutions, however, SQL solutions suffered from a high error rate in some runs of the experiments.

Future work: There are several aspects that can be improved in the future. First we aim to consider a time-series optimised database solution and compare it against MongoDB. For example, InfluxDB is a popular solution for time series data. Several other tests can also be conducted on both datasets such as functional testing and fault tolerance testing. The scalability of both solutions in terms of the deployed nodes is another point of concern. Another goal is to integrate both SQL solutions and NoSQL solutions in one system to take advantage of the ACID properties for critical data, such as payment transaction. Another future work is to optimise the storage schema and study the effects of different storage schemas. We also intend to study the effects of different privacy and security measures on the performance of MongoDB.

XII. Lessons learned

Throughout this project there were several lessons learned across the various steps conducted. The first noticeable lesson learned is the lack of transnational research for database systems in the context of the smart grid. That is, the trade-offs of several design choices for database systems are not quantitatively reported in on the specific applications. To highlight this point more, in machine learning researches, the algorithms reported in the literature are studied in smart grid context to better help researchers and practitioners understand the specific trade-offs when applied to their respective field, for example, how to study the nature of the data, what are the best preprocessing techniques for this type of data, what is considered an acceptable results, comparison between the ease of implementation and performance, or what is the best evaluation metric and procedures to be used. All these concerns can be intimidating and hard to grasp for practitioners in the field that are willing to take advantage of the technology. Another lesson was that while setting up the SQL database and experimenting with different attributes and different schema designs, we faced several issues and it was extremely difficult and time consuming in comparison to NoSQL solutions. This is due to the ACID constraints in SQL, which taught us the importance of planning the testing procedure in SQL databases beforehand. Another lesson learned was the lack of support from the testing community on both MongoDB and Jmeter. It was extremely time consuming to debug the errors that aroused and in many cases we had to refer to the documentations for debugging. On the other hand, MySQL testing was easier, well-supported by the testing community, and debugging was easy. Debugging MongoDB required extensive knowledge of the testing tool and Java. As an example, we faced an issue that MongoDB connections weren't automatically closed when

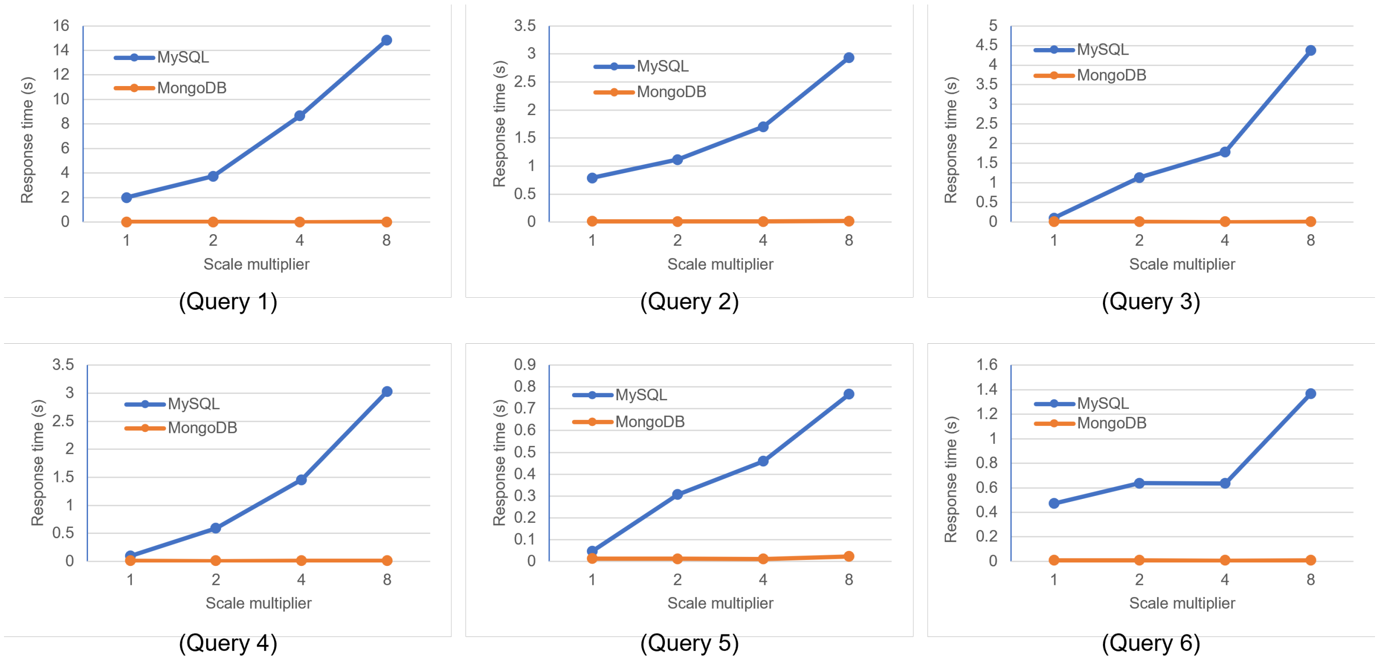

Fig. 5. Comparison between MySQL and MongoDB in database scalability

queries were finished. MongoDB automatically closed the sockets when it was overwhelmed leading to the high error rate. This problem was solved by closing all connections, which surprisingly was explicitly mentioned to only be used as a last resort in the MongoDB API documentation.

Acknowledgment

The authors would like to thank...

References

[1] V. Marinakis, "Big data for energy management and energy-efficient buildings," *Energies*, vol. 13, no. 7, p. 1555, 2020.

[2] L. C. Siebert, A. R. Aoki, G. Lambert-Torres, N. Lambert-de Andrade, and N. G. Paterakis, "An agent-based approach for the planning of distribution grids as a socio-technical system," *Energies*, vol. 13, no. 18, p. 4837, 2020.

[3] K. Zhou, C. Fu, and S. Yang, "Big data driven smart energy management: From big data to big insights," *Renewable and Sustainable Energy Reviews*, vol. 56, pp. 215–225, 2016.

[4] Y. Xu and J. Wang, "Massive power information processing scheme based on mongodb," in *IOP Conference Series: Earth and Environmental Science*, vol. 440, no. 3. IOP Publishing, 2020, p. 032020.

[5] M. Ghorbanian, S. H. Dolatabadi, and P. Siano, "Big data issues in smart grids: A survey," *IEEE Systems Journal*, vol. 13, no. 4, pp. 4158–4168, 2019.

[6] A. Zainab, A. Ghrayeb, D. Syed, H. Abu-Rub, S. S. Refaat, and O. Bouhali, "Big data management in smart grids: Technologies and challenges," *IEEE Access*, vol. 9, pp. 73 046–73 059, 2021.

[7] Y. Zhang, T. Huang, and E. F. Bompard, "Big data analytics in smart grids: a review," *Energy informatics*, vol. 1, no. 1, pp. 1–24, 2018.

[8] A. Bestavros, L. Hutyra, and E. Terzi, "Scope: Smart-city cloud based open platform and ecosystem," *Boston University: Boston, MA, USA*, 2016.

[9] N. Walravens and P. Ballon, "Platform business models for smart cities: From control and value to governance and public value," *IEEE Communications Magazine*, vol. 51, no. 6, pp. 72–79, 2013.

[10] I. Vilajosana, J. Llosa, B. Martinez, M. Domingo-Prieto, A. Angles, and X. Vilajosana, "Bootstrapping smart cities through a self-sustainable model based on big data flows," *IEEE Communications magazine*, vol. 51, no. 6, pp. 128–134, 2013.

[11] S.-V. Oprea and A. Bâra, "Machine learning algorithms for short-term load forecast in residential buildings using smart meters, sensors and big data solutions," *IEEE Access*, vol. 7, pp. 177 874–177 889, 2019.

[12] B. Cheng, S. Longo, F. Cirillo, M. Bauer, and E. Kovacs, "Building a big data platform for smart cities: Experience and lessons from santander," in *2015 IEEE International Congress on Big Data*. IEEE, 2015, pp. 592–599.

[13] S.-V. Oprea, A. Bara, B. G. Tudorică, and G. Dobrița, "Sustainable development with smart meter data analytics using nosql and self-organizing maps," *Sustainability*, vol. 12, no. 8, p. 3442, 2020.

[14] C. Costa and M. Y. Santos, "Reinventing the energy bill in smart cities with nosql technologies," in *Transactions on engineering technologies*. Springer, 2016, pp. 383–396.

[15] E. N. Yilmaz, H. Polat, S. Oyucu, A. Aksoz, and A. Saygin, "Data storage in smart grid systems," in *2018 6th International Istanbul Smart Grids and Cities Congress and Fair (ICSG)*. IEEE, 2018, pp. 110–113.

[16] M. Arenas-Martínez, S. Herrero-Lopez, A. Sanchez, J. R. Williams, P. Roth, P. Hofmann, and A. Zeier, "A comparative study of data storage and processing architectures for the smart grid," in *2010 First IEEE International Conference on Smart Grid Communications*. IEEE, 2010, pp. 285–290.

[17] H. Chihoub and C. Collet, "A scalability comparison study of data management approaches for smart metering systems," in *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE, 2016, pp. 474–483.

[18] Y. Wang, Q. Chen, T. Hong, and C. Kang, "Review of smart meter data analytics: Applications, methodologies, and challenges," *IEEE Transactions on Smart Grid*, vol. 10, no. 3, pp. 3125–3148, 2018.

[19] K. Chen, K. Chen, Q. Wang, Z. He, J. Hu, and J. He, "Short-term load forecasting with deep residual networks," *IEEE Transactions on Smart Grid*, vol. 10, no. 4, pp. 3943–3952, 2018.

[20] S. V. Oprea, A. Bâra, and G. Ifrim, "Flattening the electricity consumption peak and reducing the electricity payment for residential consumers in the context of smart grid by means of shifting optimization algorithm," *Computers & Industrial Engineering*, vol. 122, pp. 125–139, 2018.

[21] M. Mousa, S. Abdelwahed, J. Kluss *et al.*, "Review of fault types, impacts, and management solutions in smart grid systems," *Smart Grid and Renewable Energy*, vol. 10, no. 04, p. 98, 2019.

[22] S. Khatoon, A. K. Singh *et al.*, "Effects of various factors on electric load forecasting: An overview," in *2014 6th IEEE Power India International Conference (PIICON)*. IEEE, 2014, pp. 1–5.

[23] C. Si, S. Xu, C. Wan, D. Chen, W. Cui, and J. Zhao, "Electric load clustering in smart grid: Methodologies, applications, and future trends,"

Journal of Modern Power Systems and Clean Energy, vol. 9, no. 2, pp. 237–252, 2021.

[24] “indexes in mongodb manual.” [Online]. Available: https://www.mongodb.com/docs/manual/indexes

APPENDIX A
MONGODB QUERIES

```
QUERY #1
db.collection('billing').aggregate([
{
$match: {
consumer_id: new ObjectId(testIdC),
month: 'August',
year:'2020'
}
},
{

$lookup: {
from: 'consumption',
// localField: 'consumer_id',
// foreignField: 'consumer_id',
as: 'consumption',
let: { consumer_id: '$consumer_id'},
pipeline: [
{
$match: {
   $expr: {
 $and: [
{ $eq: ['$consumer_id', '$$consumer_id']},
{ $gte: [ '$timestamp', "2020-08-01 00:00:00" ] },
{ $lte: [ '$timestamp', "2020-09-01 00:00:00" ] }
]
   }
}
},
]
}

},

{
$project: {
// _id: '$consumer_id',
consumer_id: 1,
month: 1,
year: 1,
amount_due: 1,
consumption: { $sum: '$consumption.consumption' }
}
}

])



QUERY #2
db.collection('billing').updateOne({
consumer_id: new ObjectId('635d7197aaa1c9a9703b8e2c'),
month: 'August',
year: 2020,
}, {
$set: {
paid: true,
amount_due: 0.0,
}
})



QUERY #3
db.collection('consumption').aggregate([
{
$match: {
timestamp: {
$gt: '2020-01-01 00:00:00',
$lt: '2021-01-01 00:00:00'
}
}
},
{
$lookup: {
from: "consumer",
localField: "consumer_id",
foreignField: "_id",
as: "consumer"
}
},
{
$lookup: {
from: "zone",
localField: "consumer.zone_id",
foreignField: "_id",
as: "zone"
}
},

{
$group: {
_id: "$zone.name",
total: {
'$sum': '$consumption'
}
}
},
])
```

```
QUERY #4
db.collection('zone').aggregate([
{
$lookup: {
from: "consumer",
localField: "_id",
foreignField: "zone_id",
as: "consumer"
}
},
{
$lookup: {
from: "consumption",
let: { consumer_idx: "$consumer._id" },
pipeline: [
{
$match: {
$expr: {
$and: [
{ $in: ['$consumer_id', '$$consumer_idx'] },
{ $gt: ['$timestamp', '2020-01-01 00:00:00'] },
{ $lt: ['$timestamp', '2021-01-01 00:00:00'] },
]
}
}
}
],
as: "consumption"
}
},
{
$lookup: {
from: 'zonetemp',
let: { zone_id: "$_id" },
pipeline: [
{
$match: {
$expr: {
$and: [
{ $eq: ['$zone_id', '$$zone_id'] },
{ $gt: ['$timestamp', '2020-01-01 00:00:00'] },
{ $lt: ['$timestamp', '2021-01-01 00:00:00'] },
]
}

// 'timestamp': { $gt: '2004-01-01 00:00:00', $lt: '2009-01-01 00:00:00'}
}
}
],
as: "zonetemp"
}
},
{
$project: {
_id: "$name",
total_consumption: {
'$sum': '$consumption.consumption'
},
avg_temp: {
'$avg': '$zonetemp.temperature'
}
},
},
])



QUERY #5
db.collection('consumption').find({ consumption: 0.0 })


QUERY #5
db.collection('consumption').insertOne({
timestamp: '2003-07-09 17:32:44',
consumption: 44.83234,
consumer_id: new ObjectId("6353e4f19562b54233661b45"),
remove: true
})
```

APPENDIX B
QUERY PLAN AND COST ANALYSIS

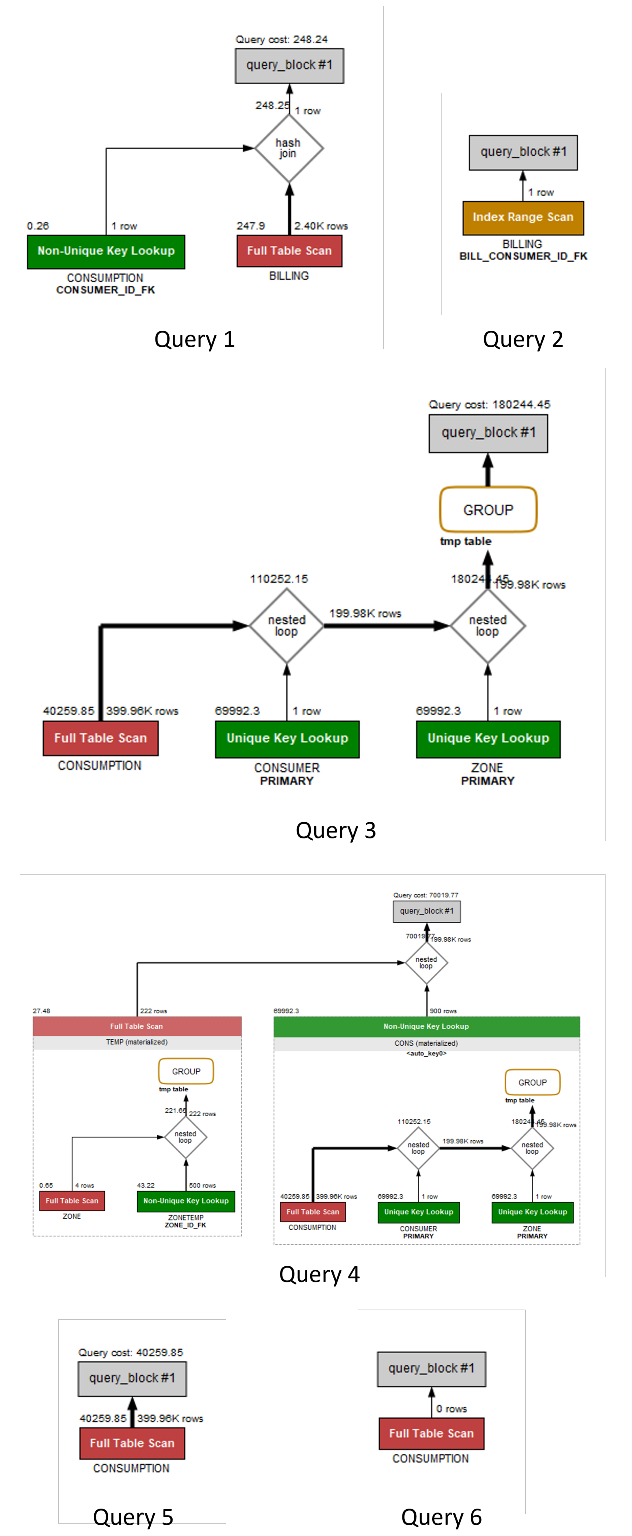

Fig. 6. Query plan and cost analysis