Towards a Traffic Accident Database in a Smart-City Use Case

Semester Assignment for the Course 'Modern Database Systems'
Master of Science for the Course of Study 'Digital Sciences'
Faculty of Computer Science and Engineering Science
of the TH Köln – University of Applied Sciences

Submitted by :	Andreas Kruff & Anh Huy Matthias Tran
Matriculation Number:	11135772 & 11134736
Lecturer:	Prof. Dr. Johann Schaible

Cologne, 28.06.2023

Kurzfassung/Abstract

The paper inspects the performance of SQL and NoSQL databases on the use case of a large-scale traffic accident database fed with live data from Switzerland, where analysts automatically create monthly traffic reports based on five key queries. For this, the paper proposes the usage of the Cassandra Wide-Column Database in order to achieve high, columnar aggregate performance for specific application queries in order to achieve a consistent writing performance while still providing superior query performance for the key queries. Insights from the experiments suggest that Oracle achieves superior writing performances due to its row-based nature, while Cassandra manages to outperform Oracle on three out of five pre-defined key queries. Additional consistency tuning measures via consistency levels improve Cassandra's read consistency at the cost of query latency.

Keywords: Database, Relational Database, NoSQL Database, Wide-Column Store, Cassandra, Traffic Analysis, Switzerland

Contents

List of Tables

List of Figures

Introduction

Over the development of the last two decades, a variety of new type of databases have emerged in the form of NoSQL databases[18]. These were developed in order to properly deal with increasing challenges in terms of volume, variety, value, velocity and veracity[10]. These NoSQL databases span from wide-column stores, enabling superior performance in aggregating values and graph databases allowing a more intuitive way in displaying relations between entities in a more digestive manner[4][20]. Despite their own fair share of advantages, however, usage of relational database management systems(RDBMS) remains more common than the usage of their newer NoSQL counterparts[19].
This paper aims to inspect the performance of the relational database 'Oracle' with the performance of the NoSQL wide-column store database 'Cassandra' on the specific use case of creating in-depth monthly analysis reports about traffic accidents in a smart city context, where live data is continuously fed into the database. The repository of this paper can be found in the provided GitHub Repository[1].

1 Use Case: Smart City Traffic Accident Database

The paper's use case describes a traffic accident database continuously collecting live traffic data from local authorities in various municipalities and cantons in Switzerland. The database is then used by their respective data analyst to create monthly traffic reports in order to gain further insight into traffic accidents and possibly related attributes such as the time of accident, vehicle mix and pedestrian involvement. Future endeavors of this database include the addition of new sensor data inspired by the Basel Smart City Approach[8], where additional contextual data such as vehicle speed are continuously recorded, with new types of categories emerging as the need for information requires[7].

1.1 Data & Use Case Requirements

The data contains 214049 records on traffic accidents, spanning from 2011 to 2022 in Switzerland[9]. It contains the following relevant information which can be grouped up as followed:

- **Accident Data:** [Accident Type], [Accident Severity], [Involvement of Pedestrians/Bicycles/Motorcycles], [Road Type]
- **Geolocation Data:** [Municipality Code], [Canton Code]
- **Temporal Data:** [Year], [Month], [Day], [Weekday], [Hour]

As such, the **requirements** of this use cases can be defined as follows:

Consistent Writing Operations While the data does not change very often after initial data entry outside of retroactive schema changes, the database continuously gathers data from live sensors from various kinds of cantons and municipalities, leading to a need for consistent writing operations in an analysis use case.

[1] https://github.com/AH-Tran/MDS_SmartCity

Low Latency Read Operation for Key Queries Under the use case of creating periodic, monthly reports, analysts utilize **pre-defined key queries** that are essential for each monthly report. As such, it is important to create a database that can deliver low latency read operations for these specific **key queries**.

CAP & PACELC Theorems As described in the CAP Theorem, most databases can at most prioritize two of the relevant consistency, availability and partitioning concepts, with most NoSQL databases being either consistent and partition tolerant(CP) or available and partition tolerant(AP)[2]. An extension to this is the PACELC Theorem, describing a database's priority in case of network partitioning or a faultless run[1]. In this use case, as the platform of a monthly analysis, the database does not have provide 100 % availability in the same manner an e-commerce or video hosting site would have to prioritize. Access to the database is mostly required in monthly intervals for the creation of the monthly report, and as such, the use case would require a mostly consistent system(CP), with the databases prioritizing consistency over availability in the case of partitioning failure and consistency over latency in a normally running operation

1.2 Choosing a NoSQL Database

When considering the aforementioned requirements, the wide-column store database **Cassandra** was chosen.
Cassandra is a popular NoSQL distributed database based on the wide-column store paradigm. By its nature, it focuses on horizontal scalability, replication, fault tolerance and a flexible approach to schema definition. This is done by allowing the pluggable creation of clusters, nodes and 'vnodes' in order to organize data sets via partitions keys[14]. Prominent examples for the usage of Cassandra can be found with IBM's Global Mailbox Client in the Sterling B2B Integrator, which is used as a tool for managing business processes and the exchange of business data securely and efficiently[12], a Cassandra use case in order to geographically distribute the nodes on multiple datacenters around the world without effects to its throughput and [17] and Netflix with its asset management service[16].

For this paper's use case, Cassandra was chosen for the following reasons:

- **Consistent Writing Operations:** While Cassandra innately prioritizes availability over consistency, the database offers various tuning options such as quorum mechanisms in order to improve consistency at the expense of availability.
- **Low Latency Read Operations for Key Queries:** Cassandra works under the principle of 'Query-driven Modeling', where each key query is supported by a specific table for the purpose of retaining high write throughput while achieving high read performance for the specified key queries through the means of primary and partition keys. This is done at the expense of introducing denormalization and data redundancy into the database. Additionally, as a wide-column store database, it is exceptionally suited for calculating aggregates across specified columns, making it ideal for the analysis report use case.

- **CAP & PACELC:** While Cassandra describes themselves as a database prioritizing availability over consistency(AP), it offers tunable consistency in the form of requiring a certain level of quorum for reading operations and customizable replication factors for writing operations. These options can be selected at the expense of Cassandra's innate availability, making it possible to create a database aligning with the CP paradigm.
- **Scalability:** As a NoSQL database not tied to a specific schema, Cassandra supports the concept of cluster, nodes and 'vnodes' in order to horizontally scale a database as new relevant topics and needs emerge[13]. This is especially relevant in the context of a smart city, where the city is continuously equipped with new types of sensors and data, requiring new tables and new key queries.

2 Methods & Approaches

This section will shortly describe the steps that have been made in terms of data modeling in order to properly ingest and organize the data for the relational database Oracle and the wide-column store database Cassandra.

2.1 Oracle: Data Modeling

In order to properly transform the data for the relational database, the paper proposes following the principles of data normalization according to the 3 normal forms(NF).

- **1NF:** Already given in the provided data set, as all columns only contain single values
- **2NF:** Already given, as the dataset only contains one primary key in the form of 'AccidentID'
- **3NF:** In order to avoid transitive dependencies, the columns [RoadType], [AccidentType], [AccidentSeverityCategory] and [CantonCode] containing codes were set as foreign keys, while the corresponding description columns of the codes were distributed to an additional table for the purpose of assigning the descriptions to their relevant codes.

The resulting data schema can be seen in Figure 1.

2.2 Cassandra: Defining Application Queries

As Cassandra follows a query-driven approach in order to optimize the query performance for key queries, it is important to define these key queries as application queries, that are central for designing the later tables. As the use case demands monthly analytic reports about traffic accidents in different cantons on a data set that spans over more than 10 years, with a focus on involvement statistics such as pedestrians, bicycles and motorcycles, the queries can be defined as follows sorted by their granularity and complexity:

- **[Q0]**: Query for a specific accident by [ID]

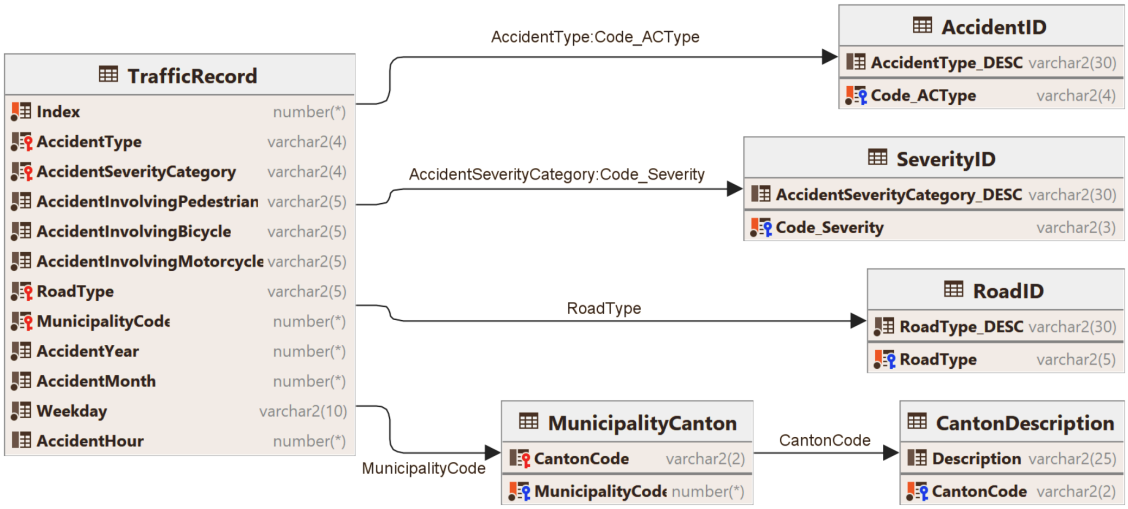

Figure 1 Oracle Schema of the Traffic Accident Database

- **[Q1]**: Query for [all accidents] in a [specific canton] and a [specific year]
- **[Q2]**: Query for the involvement of [bicycles], [pedestrians] or [motorcycles] in [traffic accidents]
- **[Q3]**: Query to compare [road types] based on the [amount of accidents] (Optionally filtered by [year], Involvement of [X] ...)
- **[Q4]**: Query to identify the [severity] of the accidents for the different [parties involved] in a given [year]

The complete SQL/CQL declaration of each query can be found in Appendix (A) and (B).

2.3 Cassandra: Data Modeling

Due to the use case featuring a data set spannning over more than 10 years, the decision was made to mainly use the column [AccidentYear] as partition key and the columns [AccidentMonth] and [AccidentID] as the clustering key for efficient searching and grouping within the data set in the monthly report. For specific key queries, additional columns are added as either partition or clustering key depending on the use case. Additionally, secondary indexes are introduced where appropriate. The resulting data schema can be seen in Figure 2.

The reasoning behind this schema can be elaborated as follows:

- **[Q0]**: Using [AccidentID] as composition key leads to small partitions and optimal balancing between the nodes in the cluster, while not allowing any WHERE Conditions without huge loss in efficiency.
- **[Q1]**: Allows analysis of specific accident IDs separated with [AccidentYear] as their respective partition key.

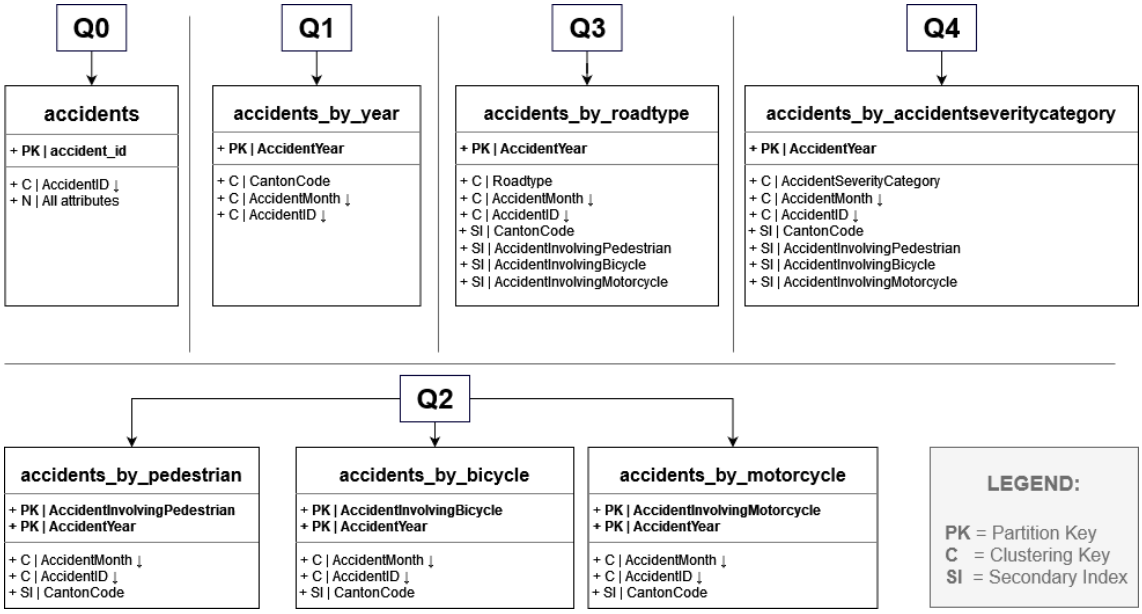

Figure 2 Cassandra Data Schema for Key Queries 0-4

- **[Q2]**: The data was split into [three separate tables] to allow direct search operations on the involvement of either [pedestrians], [bicycles] or [motorcycles] as partition keys in traffic accidents.
- **[Q3]**: Focus on identifying pattering regarding the amount of accidents depending on the [roadtype] as clustering key.
- **[Q1]**: Focus on analyzing the resulting [severities] of traffic accidents as clustering key according to their related attributes.

During the data modeling process, multiple sharding strategies were employed for different use cases. The primary objective of utilizing different sharding strategies is to enhance the efficiency of read and write operations by structuring the data in a manner that minimizes the number of partitions that need to be accessed. This includes the usage of hash sharding for [Q0] resulting in a perfect load distribution across the different nodes in the cluster. This kind of distribution is desirable in Q0, as when searching for a specific data point, a cluster-wide search becomes necessary, following the value of the hash assigned by the [accidentID] saves operation time. In contrast to [Q0], [Q2] involves the usage of sharding by entity groups, where the data is partitioned regarding the involvement of either pedestrians, bicycles or motorcycles in a given year, allowing direct access to all the required data within a single partition, while [Q1], [Q3] and [Q4] are sharded by their respective temporal range attributes. Both of these sharding strategies are specialized for querying interrelated data, at the expense of resulting in unbalanced load distribution among the shards.

3 Experimental Setup

For the sake of reproducibility and further explaining the design decisions when customizing the Cassandra database to the use case, this section explains the experimental setup that

was used for implementing the Cassandra database in an experimental docker environment in order to later utilize it to evaluate its query performance.

3.1 Node Setup

Due to Cassandra database running locally in a docker environment, the **[SimpleStrategy]** was chosen, as it is suited for the usage of only a single datacenter while keeping the complexity simple. To be able to experiment with consistency levels (CL) in the later experiments, four nodes were created within the cluster to enable the 'Quorum Concensus' with as few nodes as possible in order to keep the experiments as light-weight as possible[11]. Furthermore, the replication factor was set to the factor of three, with the four nodes divided into two writing nodes and two reading nodes, with the number of tokens randomly assigned to each node changed from the default value 256 to 16 due to reported availability concerns at higher token ranges while still allowing elasticity[15].

3.2 Table Settings

When customizing the table settings for the use case at hand, most settings were left at their default values, with the exception for certain key decisions as seen in Appendix (C) marked in green. The first key decision includes changing the **[bloom filter]**, which is a probabilistic data structure used to determine whether the file contains the searched partition or not. The documentation states that the increase in prediction accuracy is accompanied by a non-linear increase in memory usage. Therefore, the documentation suggests increasing the bloom filter(0.1 -> 0.3), thereby decreasing the prediction accuracy for analytic use cases in order to limit the memory usage[2].

In order to enable faster access to the most recent monthly data, as required by the use case, the caching parameter of each table was customized. As analysis of the data set shows that an average of 1500 accidents occur per month in Switzerland, the cache was set to store the latest 1500 rows in the cache in addition to all the keys for the purpose of enhancing reading operation at the expense of a higher incurred memory and disk usage.

An additional peculiarity of the use case is related to a Swiss legislation, requiring the removal of data after a ten-year period[3]. Due to this, the **[default_time_to_live]** of data was set to ten years(315360000 seconds). Additionally, as the data set of the use case is represented as a mostly immutable time series of traffic accident data that does not change very often after initial data entry, the **[Time Window Compaction Strategy(TWCS)]** was chosen, which is recommended for the aforementioned data set type with a predefined time-to-live(TTL).

In addition, Cassandra's documentation states, that the 'contents of an entire SSTable likely expire at approximately the same time' and can therefore be dropped completely, which results into the act of reclaiming space to be much more reliable[3].

As required by the analysis use case, in order to tune the Cassandra database more for con-

[2]https://cassandra.apache.org/doc/4.1/cassandra/operating/bloom_filters.html
[3]https://cassandra.apache.org/doc/4.1/cassandra/operating/compaction/twcs.html

sistency, the **read_repair** option was set to 'BLOCKING', therefore blocking all reading operations upon the detection of unfixed inconsistencies until the issue at hand has been repaired. This ensures that every reading operation retrieves the latest, consistent data, with the downside of higher latencies and interrupted availability in case of failure.

4 Results & Discussion

In the following section, the paper will compare the query performance of the relational database Oracle with the wide-column store Cassandra in a comparable docker environment and discuss the observation as outlined in their respective tables. Relevant metrics include disk size, writing performance upon initial setup and most importantly reading operations on the key queries.

4.1 Disk Size & Writing Operations

In the first comparison, the paper inspects the difference of resulting disc storage size between the original raw data set, the Oracle database and the Cassandra database set with four nodes and a replication-factor of 3, as well as inspecting the writing operation speed of the initial data import.

System	Size (in MB)	Writing Operation Speed
Raw Data Set	32	-
Oracle	16.05632	1.031 MB/sec
Cassandra*	152.1	0.058 MB/sec

* 4 nodes, replication_factor = 3

Table 1 Resulting Disk Storage Size in MB and Writing Operation Speed in MB/sec

As seen in Table (1), Cassandra's architecture involving the usage of nodes and the query-driven design mentioned in Section (2) leads to bigger disk sizes due to data redundancy and denormalization for the sake of supposedly improving read performance of the key queries. When observing the writing operation speed, it can be observed that Oracle as a relational, row-store database excels at quick writing operations. It shows superior performance in comparison to Cassandra, which, as a wide-column store, is innately slower at the initial data import while also having to replicate at the factor three across its nodes.

4.2 Query Performance & Reading Operations

When comparing reading operations between Oracle and Cassandra, the decision was made to rely on the respective query latency of each query as outlined in Section (2), as Cassandra offered no approachable way to calculate theoretical I/O costs of a query without imposing a performance penalty through tracing.
As such, in order to properly evaluate the query performance on reading operations, each query from [Q0] to [Q4] were executed 10 times with the cache cleared and disabled after each query execution and then subsequently averaged out in order to account for outliers during execution, with the resulting query latency including both the respective execution

time and fetching time.

In Cassandra's use case, the following levels of consistencies (CL) were tested in order to evaluate their impact on querying speed, in descending order from *[most consistent with the most latency]* to *[least consistent with the least latency]*: [ALL], [QUORUM], [ONE]

System	**Query**	**Consistency Level (CL)**	**Latency (in ms)**	Increase (in %)
Oracle	Q0	-	44	-
Cassandra	Q0	ONE	63	+ 43.18
Cassandra	Q0	QUORUM	97	+ 120.45
Cassandra	Q0	ALL	87	+ 97.72
Oracle	Q1	-	100	-
Cassandra	Q1	ONE	46	- 54
Cassandra	Q1	QUORUM	127	+ 27
Cassandra	Q1	ALL	111	+ 11
Oracle	Q2	-	123	-
Cassandra	Q2	ONE	97	- 21.14
Cassandra	Q2	QUORUM	184	+ 49.59
Cassandra	Q2	ALL	147	+ 19.51
Oracle	Q3	-	105	-
Cassandra	Q3	ONE	67	- 36.19
Cassandra	Q3	QUORUM	341	+224.77
Cassandra	Q3	ALL	289	+ 175.24
Oracle	Q4	-	77	-
Cassandra	Q4	ONE	78	+ 1.29
Cassandra	Q4	QUORUM	91	+ 18.19
Cassandra	Q4	ALL	81	+ 5.19

Table 2 Query Latency in Milliseconds with the best Performance marked green (grouped by Database, Query and Consistency Level)

When observing Table (2), it becomes apparent, that when compared at the same consistency level, both Oracle and Cassandra achieve very similar query latency for [Q4] at 77ms for Oracle and 78ms for Cassandra, while Oracle achieves a superior performance for [Q0] at 44ms when compared to Cassandra at 63ms. Oracle's superior performance at [Q0] could be due to the nature of the query, which involves a simple lookup of a specific accident's [AccidentID] across a large table, making the scan process through the [AccidentID] as a primary key a very easy task for a relational database, whereas for Cassandra, the data is partitioned across the 11-year span of the dataset, making whole dataset scans more costly.Additionally, the queried ID('112') results in Cassandra following a random hash according to the partition key, while for Oracle, it simply involves following the natural order of IDs. In Oracle's case, the low ID number is even more beneficial. In contrast to that, it can be observed that Cassandra at consistency level 'ONE' achieves superior query performance for [Q1], [Q2] and [Q3] when compared with Oracle. This could in part be explained due to the query-driven approach mentioned in Section (2), where Cassandra's Table Schema is mostly designed around the specific queries for the purpose of optimizing their specific query performance, avoiding costly queries as often seen in complicated SQL statements in relational databases, at the expense of data redundancy.

When comparing the query performance of Cassandra across different consistency level, it expectedly becomes apparent that the least consistent level ‘ONE‘ offers the lowest query latency, with more consistent levels such as ‘QUORUM‘ and ‘ALL‘ provideing a query latency increase for the sake of higher consistency when reading at least two replicas in the case of ‘QUORUM‘ and all four nodes in the case of ‘ALL‘. An interesting observation emerges when the query latency values between ‘QUORUM‘ and ‘ALL‘ are compared with each other. Despite ‘ALL‘ requiring all replicas of a data point to respond and ‘QUORUM‘ only the quorum of a maximum of three replicas, ‘ALL‘ displays lower query latency than ‘QUORUM‘. This, however, is upon further research not an unknown phenomenon, where official documentation describes situations where consistency levels can affect performance to a degree where ‘ALL‘ performs better than ‘QUORUM‘ in cases where the read operation deals with smaller data sets and where node performance is relatively equal with no slow nodes involved[6].

5 Future Outlook: Further NoSQL Database Techniques

This section will elaborate on appropriate NoSQL concepts and techniques that could be applied, where the database would be deployed in a real world production environment, where horizontal scalability, replication, failure protocols and back up procedures are required to be considered[5].

The first relevant consideration concerns itself with the topic of **Schema Evolution** and **Data Migration** under the context of the smart city use case. A quick observation of Basel’s example of a smart city project shows new data sets emerging on a regular basis, with 157 data sets dated to the most current month ‘June 2023‘ alone, containing differing degrees of fast-moving or stationary data, accompanied by the total amount of tables increasing from 230 to 234[4]. Additionally, new sensor technology can also dynamically change the amount and types of capturable data, therefore requiring a flexible, ideally zero-downtime database schema capable of being adapted on. In order to answer this use case requirement, future development should include forward engineering strategies in order to keep the schema up-to-date.

On the other hand, the monthly analysis aspect of the use case does not require immediate action when it comes to **Data Migration**, as older datasets do not contain newly introduced column values in any case and at most, two most recent months are mostly of concern. Therefore, a **Lazy Strategy** approach might be sufficient for the use case, with additional advantages including improved availability through decreased downtime. Additionally, the **Predictive Migration** might be another alternative, considering that the database is partly continuously fed with time-sensitive sensor data that is ideally processed in real time. As such, it should realistically be possible, to predict database usage depending on past trends when it comes to creating monthly reports, such as certain key queries only occurring during certain month periods.

For the **Replication Strategy**, the use case requires a strategy that ideally keeps replicas consistent. For this, the synchronous eager migrating might be a better fit for this use case

[4](https://basel-stadt.opendatasoft.com/)

than the asynchronous (lazy) migration replication, as it minimizes the risk of data loss while providing a high consistency across all replicas.
Cassandra's general architecture follows the **Multi-Master Model** where most nodes have similar responsibilities in opposition to the regular Master-Slave Model. While this structure supports fast and highly available reads, it might require the implementation of coordination protocols such as 'Paxos' in order to support better consistency that is required by the use case, further increasing the project's complexity and with that, potential new avenues for failure. Additionally, the organization between the clusters should in a real production environment adopt the 'NetworkTopologyStrategy', in order to orchestrate proper replication across multiple clusters and datacenters. In order to improve general query performance of the database, there exists a trade-off between local secondary indexing and global secondary indexing. While global secondary indexing improves the query performance in terms of reading operations, due to only nodes containing results needing to be queried, local secondary index offers better writing performance and increased consistency by not following the consistent index-maintenance introduced through global secondary indexing, making it possibly advantageous for the purpose of processing live sensor data.
Lastly, in order to enhance the querying on single columns, utilizing early materialization(EM) might be suitable for the use case, as the set key queries for the monthly traffic accident reports should not differ greatly from month to month. An implementation of EM would enable the preprocessing of the query process when it comes to selecting and aggregating often scanned data, similar to the concept of caching the latest 1500 rows in Section (3). For this, further considerations concerning the 'MemTable' and 'Commitlog' settings in Cassandra have to be made in order to store the aggregated data in-memory more appropriately for the monthly use case.

6 Conclusion

As shown in the aforementioned results, it can be seen that despite tuning Cassandra's initial settings towards consistency at the expense of availability and latency, that it manages to outperform Oracle in [Q1], [Q2] and [Q3], while performing similarly at [Q4] and being outperformed by Oracle at [Q0]. When tuning Cassandra to even more consistency on the query level by setting the consistency level to either 'ALL' or 'QUORUM', Cassandra does not manage to outperform Oracle in any key query when it comes to query latency. This performance decrease in latency, however, comes with the other advantages inherent to Cassandra's design, that being inherently fault-tolerant and horizontally scalable through the implementation of clusters, nodes and 'vnodes', making it suitable for a changing and growing analysis database for live traffic and city data, where new topics with new sensor data can continue to be added dynamically to the flexible wide-column store data schema. While the data set in its current inception is relatively small, implementing this paper in a more realistic environment with extreme data volume might lead to more significant differences in query run time, which should be next in line for future research.

References

[1] Abadi, D. (2012). Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. *Computer*, 45(2):37–42.

[2] Apache (2023). Guarantees | Apache Cassandra Documentation.

[3] Bundesamt für Strassen (ASTRA) (2023). Verordnung über das informationssystem strassenverkehrsunfälle. Art. 9.

[4] Corbellini, A., Mateos, C., Zunino, A., Godoy, D., and Schiaffino, S. (2017). Persisting big-data: The nosql landscape. *Information Systems*, 63:1–23.

[5] DataStax (2020). Planning and testing apache cassandra deployments. last accessed: 28.06.2023.

[6] DataStax (2022). How consistency affects performance | CQL for Cassandra 3.0.

[7] Fachstelle für Open Government Data (2023a). Datenportal BS.

[8] Fachstelle für Open Government Data (2023b). Welcome to Smart City Basel.

[9] GEO.ADMIN.CH (2023). Verfügbare datensätze. last accessed: 28.06.2023.

[10] Hewage, T., Halgamuge, M., Syed, A., and Ekici, G. (2018). Review: Big data techniques of google, amazon, facebook and twitter. *Journal of Communications*, 13:94–100.

[11] Huang, X., Wang, J., Qiao, J., Zheng, L., Zhang, J., and Wong, R. K. (2017). Performance and Replica Consistency Simulation for Quorum-Based NoSQL System Cassandra. In van der Aalst, W. and Best, E., editors, *Application and Theory of Petri Nets and Concurrency*, Lecture Notes in Computer Science, pages 78–98, Cham. Springer International Publishing.

[12] IBM (2023). IBM Documentation Apache Cassandra.

[13] Kuhlenkamp, J., Klems, M., and Röss, O. (2014). Benchmarking scalability and elasticity of distributed database systems. *Proceedings of the VLDB Endowment*, 7(12):1219–1230.

[14] Lakshman, A. and Malik, P. (2010). Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40.

[15] Lynch, J. and Synder, J. (2018). Cassandra availability with virtual nodes.

[16] Meenakshi, J. (2022). Data Reprocessing Pipeline in Asset Management Platform @Netflix.

[17] Ploetz, A. (2023). Two Reasons Why Apache Cassandra Is the Database for Real-Time Applications.

[18] Roy-Hubara, N. and Sturm, A. (2020). Design methods for the new database era: a systematic literature review. *Software and Systems Modeling*, 19(2):297–312.

[19] solid IT gmbh (2023). Db-engines ranking. last accessed: 28.06.2023.

[20] Zafar, R., Yafi, E., Zuhairi, M. F., and Dao, H. (2016). Big data: The nosql and rdbms review. In *2016 International Conference on Information and Communication Technology (ICICTM)*, pages 120–126.

Appendix

Appendix A Key Queries 0-4 as CQL Commands

Q0: Query for a specific accident by ID

```
SELECT * FROM traffic_accidents.accidents WHERE accidentid = 37034;
```

Q1: Query for all accidents in a specific canton and a specific year

```
SELECT *
FROM traffic_accidents.accidents_by_year
WHERE accidentyear = 2012
 AND cantoncode = 'ZH';
```

Q2: Query for the involvement of bicycles, pedestrians or motorcycles in accidents

```
-- Q2.1: Involvement of Pedestrians

SELECT *
FROM traffic_accidents.accidents_by_pedestrian
WHERE cantoncode = 'ZH'
  AND accidentyear = 2012
  AND accidentinvolvingpedestrian = true;

-- Q2.2: Involvement of bicycles

SELECT *
FROM traffic_accidents.accidents_by_bicycle
WHERE cantoncode = 'ZH'
  AND accidentyear = 2012
  AND accidentinvolvingbicycle = true;

-- Q2.3: Involvement of motorcycles

SELECT * FROM traffic_accidents.accidents_by_motorcycle
         WHERE cantoncode = 'ZH'
           AND accidentyear = 2012
           AND accidentinvolvingmotorcycle = true;
```

Q3: Query to compare road types based on the amount of accidents (Optionally filtered by year, Involvement of X ...)

```
-- Q3.1: Using Table 3.1 requires ALLOW FILTERING
SELECT roadtype, COUNT(*) AS RoadTypeCount
FROM accident_by_roadtype
WHERE accidentinvolvingbicycle = true AND accidentyear = 2013
GROUP BY roadtype,accidentyear
ALLOW FILTERING ;

-- Q3.2: Using Table 3.2 without ALLOW FILTERING

SELECT roadtype, COUNT(*) AS RoadTypeCount
FROM accident_by_roadtype
WHERE accidentinvolvingbicycle = true AND accidentyear = 2013
GROUP BY roadtype;
```

Q4: Query to identify the severity of the accidents for the different parties involved in a given year

```
SELECT COUNT(*) as bicycle_count
FROM traffic_accidents.accident_by_AccidentSeverityCategory
WHERE accidentyear = 2013 AND AccidentSeverityCategory = 'as2' AND
   ↪ accidentinvolvingmotorcycle =true;
```

Appendix B Key Queries 0-4 as SQL Commands

Q0: Query for a specific accident by ID

```
SELECT TR.*, AT."AccidentType_DESC", RI."RoadType_DESC", SI."
   ↪ AccidentSeverityCategory_DESC",CD."Description"
FROM "TrafficRecord" TR
JOIN "AccidentID" AT ON TR."AccidentType" = AT."Code_ACType"
JOIN "RoadID" RI ON TR."RoadType" = RI."RoadType"
JOIN "SeverityID" SI ON TR."AccidentSeverityCategory" = SI."
   ↪ Code_Severity"
JOIN "MunicipalityCanton" MC ON TR."MunicipalityCode" = MC."
   ↪ MunicipalityCode"
JOIN "CantonDescription" CD ON MC."CantonCode" = CD."CantonCode"
WHERE TR."Index" = 112;
```

Q1: Query for all accidents in a specific canton and a specific year

```
SELECT *
FROM "TrafficRecord" TR
JOIN "MunicipalityCanton" MC ON TR."MunicipalityCode" = MC."
   ↪ MunicipalityCode"
WHERE "AccidentYear" = 2012 AND MC."CantonCode" = 'ZH';
```

Q2: Query for the involvement of bicycles, pedestrians or motorcycles in accidents

```
SELECT *
FROM "TrafficRecord" TR
JOIN "MunicipalityCanton" MC ON TR."MunicipalityCode" = MC."
   ↪ MunicipalityCode"
WHERE "AccidentYear" = 2012 AND MC."CantonCode" = 'ZH' AND "
   ↪ AccidentInvolvingPedestrian" = 'True';
```

Q3: Query to compare road types based on the amount of accidents (Optionally filtered by year, Involvement of X ...)

```
SELECT TR."RoadType" , RI."RoadType_DESC", COUNT(*) AS
   ↪ AmountPerType
FROM "TrafficRecord" TR
JOIN "RoadID" RI ON TR."RoadType" = RI."RoadType"
WHERE "AccidentYear" = 2013 AND "AccidentInvolvingMotorcycle" = '
   ↪ True'
GROUP BY TR."RoadType", RI."RoadType_DESC"
ORDER BY AmountPerType DESC;
```

Q4: Query to identify the severity of the accidents for the different parties involved in a given year

```
SELECT COUNT(*) as bicycle_count
FROM "TrafficRecord"
WHERE "AccidentYear" = 2013 AND "AccidentSeverityCategory" = 'as2'
    ↪ AND "AccidentInvolvingMotorcycle" = 'True';

-- Code to flush the Cache and the Shared Pool of the Database (
    ↪ should be executed after every query)

ALTER SYSTEM FLUSH BUFFER_CACHE;
ALTER SYSTEM CHECKPOINT;
ALTER SYSTEM FLUSH SHARED_POOL;
```

Appendix C Cassandra Settings

Parameter	Setting	Default
additional_write_policy	‘99p’	True
bloom_filter_fp_chance	0.3	False
Caching	’keys’: ’ALL’, ’rows_per_partition’: 1500	False
cdc	false	True
comment	‘’	True
compaction	TimeWindowCompactionStrategy, window-size: ’122’, window-unit: ’DAYS’	False
compression	’chunk_length_in_kb’: ’16’, ’class’: ’org.apache.cassandra.io.compress.LZ4Compressor’	True
crc_check_chance	1.0	True
default_time_to_live	315360000	False
gc_grace_seconds	864000	True
max_index_interval	2048	True
min_index_interval	128	True
read_repair	’BLOCKING’	True
speculative_retry	’99p’	True

Table 3 Cassandra Settings with changed Settings highlighted in Green