National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

# Software Architectures for integrating timeseries databases in applications managing timeseries data on relational databases.

DIPLOMA THESIS

Spyridon Alexandros Diochnos

**Supervisor:** Vassilios Vescoukis
Professor

Athens, February 2024

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

# Αρχιτεκτονικές λογισμικού για την ενοποίηση Βάσεις δεδομένα χρονοσειρών με εφαρμογές διαχείρισης δεδομένων χρονοσειρών που χρησιμοποιούν σχεσιακές Βάσεις δεδομένων

DIPLOMA THESIS

Spyridon Alexandros Diochnos

**Supervisor:** Vassilios Vescoukis
Professor

Approved by the examination committee on March 27th.

| | | |
|---|---|---|
| Vassilios Vescoukis<br>Professor | Tsoumakos Dimitrios<br>Professor | Aris Dimeas<br>Assistant Professor |

Athens, March 2024

# Περίληψη

Οι βάσεις δεδομένων παίζουν ρόλο μείζονος σημασίας στο σύγχρονο τοπίο των ψηφιακών συστημάτων. Καθώς βρίσκονται στην καρδιά του συστήματος, η σωστή επιλογή μπορεί να επηρεάσει σε σημαντικό βαθμό την απόδοση, το κόστος, την κλιμακωσιμότητα και την ευκολία στην συντήρηση. Τα τελευταία χρονιά με την ραγδαία αύξηση του όγκου των δεδομένων, έχουν αναπτυχθεί πολλά είδη NoSQL βάσεων δεδομένων. Μεταξύ αυτών βρίσκονται και οι βάσεις δεδομένων χρονοσειρών (time series databases- TSDB) με σκοπό να βελτιστοποιήσουν τις επιδόσεις σε πολλά προβλήματα στα οποία απαιτείται επεξεργασία δεδομένων χρονοσειρών, όπως στον χώρο των οικονομικών, του περιβάλλοντος και της ενέργειας.

Σε αυτή την εργασία επιχειρούμε μια συγκριτική ανάλυση αρχιτεκτονικών λογισμικού που κάνουν χρήση κάποιων βάσεων δεδομένων χρονοσειρών και μιας σχεσιακής βάσης δεδομένων - συγκεκριμένα τις InfluxDB, Apache Druid και MySQL, αντίστοιχα. Στόχος μας είναι να διερευνήσουμε τα οφέλη, τις προκλήσεις και την διαφορά στην απόδοση που προσφέρει κάθε μια. Για να αποφανθούμε του συγκεκριμένου ερωτήματος χρησιμοποιήσαμε έναν μεγάλο όγκο δεδομένων χρονοσειρών από τις αγορές ηλεκτρικής ενέργειας στις χώρες της ευρωπαϊκής ένωσης, με τα οποία χρονομετρήσαμε εγγραφές, αναγνώσεις και μεταποιήσεις στην εκάστοτε βάση και μελετήσαμε διάφορες αρχιτεκτονικές υλοποίησης ενός πλήρους συστήματος.

Η υλοποίησή που έγινε, μας επέτρεψε να παρατηρήσουμε με συγκρίσιμο τρόπο τις επιδόσεις, τη συμπεριφορά και τη χρηστικότητα κάθε συστήματος προκειμένου να οδηγηθούμε σε συμπεράσματα χρήσιμα σε όσους αντιμετωπίζουν αποφάσεις σχετικές με βάσεις δεδομένων και αρχιτεκτονικών λογισμικού για παρεμφερή συστήματα. Συμπεράναμε, ότι η Apache Druid, αλλά πρωτίστως η Influxdb, είναι σημαντικά πιο γρήγορες από την MySQL για διαχείριση χρονοσειρών και μας επιτρέπουν να χτίσουμε λιγότερο περίπλοκες αρχιτεκτονικές.

# 1. Λέξεις Κλειδιά

Βάσεις Δεδομένων, NoSQL, Χρονοσειρές, Βάσεις Δεδομένων Χρονοσειρών, Influxdb, Apache Druid, MySQL, Απόδοση, Κόστος, Κλιμακωσιμότητα, Συντήρηση, Όγκος Δεδομένων, Εγγραφές, Αναγνώσεις, Μεταποιήσεις, Αρχιτεκτονικές Υλοποίησης, Αγορές Ηλεκτρικής Ενέργειας, Ευρωπαϊκή Ένωση, Συγκριτική Ανάλυση, Συστήματα, Χρηστικότητα

# 2. Abstract

Databases play a vital role in the modern landscape of digital systems. As they lie at the heart of the system, the right choice can significantly affect performance, cost, scalability, and ease of maintenance. In recent years, with the rapid increase in data volume, many types of NoSQL databases have been developed. Among these are time series databases (TSDBs), designed to optimize the performance of timeseries management in various problems, such as in finance, environment, and energy sectors.

In this work, we undertake a comparative analysis of architectures that use selected time series databases along with a relational database—specifically, InfluxDB, Apache Druid, and MySQL, respectively. Our goal is to explore the benefits, challenges, and performance differences each offers. To address this question, we used a large volume of timeseries data from the electric energy markets in the European Union countries, with which we timed records, reads, and transformations in each database and studied various implementation architectures of a complete system.

The implementation allowed us to observe in a comparable manner the performances, behavior, and usability of each system to lead to conclusions useful for those facing decisions related to databases and software architectures for similar systems. We concluded that Apache Druid, but primarily InfluxDB, are significantly faster than MySQL for timeseries management and allow us to build less complex application architectures.

# 3. Keywords

Databases, NoSQL, Time Series, InfluxDB, Apache Druid, MySQL, Performance, Cost, Scalability, Maintenance, Data Volume, Writes, Reads, Transformations, Implementation Architectures, Electric Energy Markets, European Union, Comparative Analysis, Systems, Usability

# 4. Acknowledgements

After completing my thesis, my five-year trip to the School of Electrical and Computer Engineering (ECE) at the National Technical University of Athens (NTUA) ends. This would not be possible without my family and friends, who supported me throughout my undergraduate studies. I would like to express my gratitude to my professor and supervisor Mr. Vassilios Vescoukis, for the opportunity to work and get hands-on experience on this subject, but also for his valuable guidance, feedback, and excellent cooperation during my thesis.

Finally, I wish to extend my special thanks to all the people who were part of this journey at the National Technical University of Athens.

Athens, Nov 2023

Spyridon Alexandros Diochnos

# 5. Εκτεταμένη Περίληψη

Η ενότητα «Introduction» της εργασίας θέτει το υπόβαθρο για μια ολοκληρωμένη εξερεύνηση των Βάσεων Δεδομένων Χρονοσειρών (TSDBs), μιας εξειδικευμένης κατηγορίας βάσεων δεδομένων που έχουν βελτιστοποιηθεί για το χειρισμό δεδομένων με χρονική σήμανση. Περιγράφεται ο κεντρικός ρόλος αυτών των βάσεων δεδομένων στο σύγχρονο ψηφιακό οικοσύστημα, δίνοντας έμφαση στο αντίκτυπό τους στην απόδοση, την επεκτασιμότητα και τη συντήρηση των ψηφιακών συστημάτων. Τα τελευταία χρόνια, η αύξηση της παραγωγής ψηφιακών δεδομένων και η ανάγκη για αποτελεσματική διαχείριση των δεδομένων αυτών, έχουν φέρει τις TSDB στο προσκήνιο. Στον χώρο των επιχειρήσεων και των οργανισμών ,προκύπτουν ολοένα και περισσότερες προκλήσεις στην επιλογή του κατάλληλου τύπου βάσης δεδομένων για να ανταποκριθούν στις μοναδικές απαιτήσεις χειρισμού δεδομένων τους. Εστιάζοντας σε διαδεδομένες TSDB όπως το InfluxDB [5] και το Apache Druid, η εργασία στοχεύει να παρέχει μια συγκριτική ανάλυση αυτών των βάσεων δεδομένων έναντι των παραδοσιακών βάσεων δεδομένων SQL όπως η MySQL. Αυτή η σύγκριση είναι κρίσιμη για την κατανόηση των διακριτών πλεονεκτημάτων και των πιθανών περιορισμών των TSDB σε διάφορα σενάρια εφαρμογής. Στην εισαγωγή αναφέρεται η σημασία των δεδομένων χρονοσειρών, τα οποία χαρακτηρίζονται από τη διαδοχική φύση τους και τον κρίσιμο ρόλο του χρόνου ως πρωταρχικής διάστασης. Αυτός ο τύπος δεδομένων μπορεί να βρεθεί σε διάφορους τομείς, συμπεριλαμβανομένων των οικονομικών, της υγειονομικής περίθαλψης, των τηλεπικοινωνιών και της διαχείρισης ενέργειας. Μελετώνται εκτενώς το InfluxDB και το Apache Druid, ο αρχιτεκτονικός σχεδιασμός τους, οι περιπτώσεις χρήσης και ο τρόπος σύγκρισής τους με συμβατικές σχεσιακές βάσεις δεδομένων όπως η MySQL. Μέχρι το τέλος αυτής της ενότητας, οι αναγνώστες αναμένεται να έχουν μια σαφή κατανόηση της σημασίας των TSDB στην ψηφιακή εποχή, το σκεπτικό πίσω από την αυξανόμενη δημοτικότητά τους και τα βασικά ερωτήματα που επιδιώκει να αντιμετωπίσει η εργασία σχετικά με την απόδοση και τη δυνατότητα εφαρμογής τους σε σύγκριση με παραδοσιακά συστήματα βάσεων δεδομένων.

## Επιλογή και Σύγκριση Βάσης

Στην ενότητα αυτή εστιάζουμε στην επιλογή του InfluxDB και του Apache Druid για μια μελέτη και σύγκριση με MySQL. Καθώς έχει αυξηθεί η σημασία των TSDB, ιδιαίτερα για εφαρμογές που απαιτούν αποτελεσματικό χειρισμό δεδομένων με χρονική σήμανση. Το InfluxDB είναι γνωστό για την υψηλή του απόδοση στον χειρισμό δεδομένων χρονοσειρών. Κατατάσσεται σταθερά ψηλά στις βάσεις δεδομένων που είναι αφιερωμένες σε δεδομένα χρονοσειρών, όπως αποδεικνύεται από τις θέσεις της στις κατατάξεις DB-Engines και G2.[1] Αντίστοιχα το Apache Druid, με τη συμμετοχή του στο Ίδρυμα Λογισμικού Apache, φέρνει ισχυρή φήμη στην κοινότητα open source κώδικα, καθιστώντας το ιδανικό υποψήφιο για σύγκριση.

Το InfluxDB, που αναπτύχθηκε από την InfluxData και κυκλοφόρησε το 2013, ξεχωρίζει για την αρχιτεκτονική του στο cloud. Έχει σχεδιαστεί για να διαχειρίζεται αποτελεσματικά δεδομένα χρονοσειρών, κατάλληλο για μια σειρά εφαρμογών, όπως η παρακολούθηση, το IoT [2] και η ανάλυση σε πραγματικό χρόνο. Η τελευταία έκδοση, το InfluxDB 3.0, εισάγει μια distributed αρχιτεκτονική για υπολογισμούς με scalability και αποθήκευση και υποστηρίζει SQL και InfluxQL /Flux για queries. Οι περιπτώσεις χρήσης του είναι ποικίλες, καλύπτοντας το monitoring, την αποθήκευση δεδομένων IoT και την ανάλυση σε πραγματικό χρόνο.

Το Apache Druid, που δημιουργήθηκε από τη Metamarkets και αργότερα εισήχθη στο Ίδρυμα Apache, είναι βελτιστοποιημένο για αναλύσεις σε real-time. Η αρχιτεκτονική του συνδυάζει στοιχεία βάσεων δεδομένων χρονοσειρών, συστημάτων αναζήτησης και columnar storage, καθιστώντας το ιδανικό για interactive analytics και event-driven data. Είναι γνωστό για την οριζόντια επεκτασιμότητα του και υποστηρίζει κατανεμημένες αρχιτεκτονικές. Το Druid είναι ιδιαίτερα κατάλληλο για εφαρμογές όπως η geospatial analysis, η μηχανική μάθηση, το AI preprocessing και η ανάλυση σε πραγματικό χρόνο.

Η σύγκριση μεταξύ InfluxDB και Apache Druid είναι περίπλοκη, λαμβάνοντας υπόψη τις διαφορετικές αρχιτεκτονικές προσεγγίσεις και τις περιπτώσεις χρήσης τους. Τα δυνατά σημεία του InfluxDB βρίσκονται στη βελτιστοποιημένη αποθήκευσή του με χρήση τεχνολογιών όπως το Parquet και το Apache Arrow, η σχεδίαση χωρίς σχήμα και η εύκολη ενσωμάτωση μέσω HTTP API ή CLI.

Οι περιπτώσεις χρήσης και των δύο βάσεων δεδομένων αντικατοπτρίζουν τα αρχιτεκτονικά τους πλεονεκτήματα. Το InfluxDB υπερέχει σε σενάρια που απαιτούν υψηλή απόδοση εγγραφής και αναζήτησης, όπως η συνεχής εγγραφή δεδομένων και η άμεση αναζήτηση. Το Druid, εστιάζει περισσότερο στην ανάλυση σε πραγματικό χρόνο και στην αποθήκευση ιστορικών δεδομένων με οικονομικά αποδοτικό τρόπο.

## Η Βάση Apache Druid

Η αρχιτεκτονική του Apache Druid αναλύεται για να παρέχει πληροφορίες για το λειτουργικό του πλαίσιο. Η βάση δεδομένων διανέμεται, με μια διαμόρφωση που περιλαμβάνει διαφορετικούς τύπους κόμβων: κόμβους Master, Query και Data. Αυτή η τμηματοποίηση είναι καθοριστική για τη βελτιστοποίηση της προσβασιμότητας δεδομένων, την επεξεργασία ερωτημάτων και τη διαχείριση της αποθήκευσης δεδομένων. Η επεκτασιμότητα κάθε τύπου κόμβου είναι ένα χαρακτηριστικό που ξεχωρίζει, επιτρέποντας την προσαρμοστικότητα του συστήματος στις μεταβαλλόμενες απαιτήσεις στις δυνατότητες αποθήκευσης και αναζήτησης.

Η ευελιξία του μοντέλου δεδομένων του Apache Druid είναι ένα βασικό στοιχείο. Σε αντίθεση με τις παραδοσιακές βάσεις δεδομένων με σταθερά σχήματα (schemas), η προσέγγιση σχήματος σε ανάγνωση του Druid επιτρέπει δυναμικό χειρισμό δεδομένων και ενσωμάτωση διαφόρων μορφών δεδομένων. Το data ingestion αναπτύσσεται, παρέχοντας μια ολοκληρωμένη κατανόηση του τρόπου με τον οποίο τα δεδομένα εισέρχονται και επεξεργάζονται μέσα στο σύστημα.

Η υποβολή ερωτημάτων στο Druid είναι ευέλικτη, προσφέροντας ερωτήματα που βασίζονται σε SQL και σε Druid Native. Παρέχονται παραδείγματα μετάφρασης από τα ερωτήματα SQL σε Druid Native, αποδεικνύοντας την ευκολία με την οποία οι χρήστες μπορούν να ενσωματώσουν την υπάρχουσα τεχνογνωσία τους SQL στο περιβάλλον του Druid.

### Η Βάση InfluxDB

Το InfluxDB , που αναγνωρίζεται ως σημαντική εξέλιξη στις κλιμακούμενες βάσεις δεδομένων, εισάγεται με την τμηματοποιημένη αρχιτεκτονική του που περιλαμβάνει ξεχωριστά στοιχεία για την απορρόφηση δεδομένων, την αναζήτηση, τη συμπίεση και τη συλλογή σκουπιδιών. Αυτός ο αρχιτεκτονικός σχεδιασμός υπογραμμίζει την αποτελεσματικότητα του InfluxDB στη διαχείριση δεδομένων σε διάφορα στάδια του κύκλου ζωής του.

Το data ingestion στο InfluxDB περιγράφεται, δίνοντας έμφαση στην επεκτασιμότητα του συστήματος στον χειρισμό διαφορετικών φόρτων εργασίας δεδομένων. Αυτή η πτυχή είναι κρίσιμη για συστήματα που ασχολούνται με κυμαινόμενους όγκους δεδομένων, όπως φαίνεται σε εφαρμογές ανάλυσης σε πραγματικό χρόνο. Όσον αφορά τα ερωτήματα δεδομένων, το InfluxDB παρουσιάζει ένα καλά δομημένο querying component, βελτιστοποιημένο για να χειρίζεται υψηλούς φόρτους εργασίας ερωτημάτων, το οποίο αποτελεί βασική απαίτηση για την ανάλυση δεδομένων χρονοσειρών.

Το component συμπίεσης(compaction) δεδομένων αντιμετωπίζει την πρόκληση της διαχείρισης πολλών μικρών αρχείων. Με τη συμπίεση αυτών των αρχείων σε μεγαλύτερα, μη επικαλυπτόμενα αρχεία, το InfluxDB βελτιώνει την απόδοση ερωτημάτων του. Επιπλέον, ο μηχανισμός συλλογής σκουπιδιών(Garbage Collection) στο InfluxDB, ο οποίος διαχειρίζεται τη διατήρηση δεδομένων και την ανάκτηση χώρου, απεικονίζει την ικανότητα της βάσης δεδομένων να διατηρεί αποτελεσματικότητα και τάξη στην αποθήκευση δεδομένων.

Η διαδικασία εγγραφής δεδομένων στο InfluxDB χρησιμοποιώντας το line protocol, καλύπτεται εκτενώς. Αυτή η εστίαση σε μια φιλική προς τον χρήστη μέθοδο εισαγωγής δεδομένων αντικατοπτρίζει τη δέσμευση του InfluxDB για προσβασιμότητα. Διερευνώνται επίσης οι δυνατότητες ερωτημάτων του InfluxDB, περιγράφοντας λεπτομερώς τη χρήση τόσο της γλώσσας Flux όσο και της InfluxQL. Αυτή η προσέγγιση με δύο γλώσσες καλύπτει ένα ευρύ φάσμα προτιμήσεων και απαιτήσεων των χρηστών.

Συζητούνται παρακάτω τεχνικές όπως η επαναχαρτογράφηση δεδομένων, η ομαδοποίηση και η συγκέντρωση (data remapping, grouping, and aggregating are discussed), υπογραμμίζοντας τις δυνατότητες της βάσης δεδομένων στη βελτιστοποίηση της χρήσης δεδομένων. Η συμπερίληψη προηγμένων λειτουργιών όπως η περιστροφή και η μείωση δειγματοληψίας, μαζί με τη δυνατότητα αυτοματοποίησης εργασιών επεξεργασίας δεδομένων, υπογραμμίζει την ολοκληρωμένη εργαλειοθήκη διαχείρισης και ανάλυσης δεδομένων του InfluxDB.

## Περιγραφή του προβλήματος

Η εργασία εστιάζει στις προκλήσεις που αντιμετωπίζει η διαχείριση δεδομένων ενέργειας μεγάλης κλίμακας με παραδοσιακές βάσεις δεδομένων SQL. Θέτει το έδαφος για μια έρευνα σχετικά με το εάν οι βάσεις δεδομένων χρονοσειρών (TSDB) μπορούν να προσφέρουν μια πιο αποτελεσματική εναλλακτική λύση, ειδικά λαμβάνοντας υπόψη τα μοναδικά χαρακτηριστικά των ενεργειακών δεδομένων, τα οποία είναι ογκώδη και ευαίσθητα στο χρόνο.

Γίνεται εμβάθυνση στα συγκεκριμένα χαρακτηριστικά του συνόλου ενεργειακών δεδομένων, τονίζοντας την πολυπλοκότητα που είναι εγγενής στον χειρισμό αυτού του τύπου δεδομένων χρησιμοποιώντας βάσεις δεδομένων SQL. Συζητά τις προκλήσεις στην επεξεργασία τέτοιων δεδομένων, συμπεριλαμβανομένων ζητημάτων που σχετίζονται με την συμπλήρωση κενών, τη συγκέντρωση δεδομένων και τις αναποτελεσματικές επιδόσεις στις βάσεις δεδομένων SQL όταν ασχολούμαστε με μεγάλα σύνολα δεδομένων με χρονική σήμανση.

Η μεθοδολογία για τη συγκριτική ανάλυση περιγράφεται σε αυτή την ενότητα. Περιγράφει τη ρύθμιση πανομοιότυπων διακομιστών για τη δοκιμή της απόδοσης της SQL έναντι των TSDB υπό ελεγχόμενες συνθήκες. Η ενότητα εξηγεί τον σχεδιασμό διαφόρων σεναρίων δοκιμών, με διαφορετικές καταστάσεις των βάσεων δεδομένων (άδειες, μερικώς γεμάτες, πλήρως φορτωμένες) και πώς αυτές οι συνθήκες επηρεάζουν την απορρόφηση δεδομένων και τους χρόνους απόκρισης ερωτημάτων.

Παρουσιάζεται μια σύγκριση των SQL και TSDB, εκτελώντας πανομοιότυπες λειτουργίες και στα δύο περιβάλλοντα. Η εστίαση είναι στην αξιολόγηση του τρόπου με τον οποίο κάθε βάση δεδομένων χειρίζεται το data ingestion και τα ερωτήματα σε διάφορους όγκους δεδομένων, παρέχοντας μια ποσοτική βάση για τη σύγκριση των επιδόσεων τους.

Αξιολογείται η λειτουργική πολυπλοκότητα που σχετίζεται με κάθε τύπο βάσης δεδομένων και στοχεύει να καθορίσει εάν τα TSDB μπορούν να απλοποιήσουν τις διαδικασίες χειρισμού δεδομένων και να αυξήσουν τη λειτουργική αποτελεσματικότητα. Αντιπαραβάλλει τις πρόσθετες απαιτήσεις δέσμης ενεργειών και μετασχηματισμού δεδομένων στην SQL με την πιο βελτιωμένη προσέγγιση που προσφέρουν τα TSDB.

Επιπλέον συζητούνται τα ευρήματα από τη συγκριτική μελέτη, με έμφαση στο εάν τα TSDB είναι πιο αποτελεσματικές από τις βάσεις δεδομένων SQL. Η ενότητα διερευνά τις επιπτώσεις αυτών των αποτελεσμάτων, στο πλαίσιο της λειτουργικής αποτελεσματικότητας και της πιθανής εξοικονόμησης κόστους στον ενεργειακό τομέα και σε άλλους κλάδους που ασχολούνται με δεδομένα μεγάλης κλίμακας.

## Οι μετρήσεις τηςMySQL

Η ενότητα «MySQL Baseline» παρουσιάζει μια κριτική ανάλυση της MySQL, που χρησιμοποιείται στο πλαίσιο της διαχείρισης δεδομένων μεγάλης κλίμακας, με χρονική σήμανση. Ο στόχος είναι να δημιουργηθεί μια βασική γραμμή απόδοσης για την MySQL.

Η μεθοδολογία περιλαμβάνει τη διαμόρφωση μιας βάσης δεδομένων MySQL ώστε να αντικατοπτρίζει ένα πλήρως φορτωμένο σύστημα, που καλύπτει 120 μήνες ιστορικών δεδομένων. Αυτή η ρύθμιση διασφαλίζει ότι οι μετρήσεις απόδοσης που προκύπτουν είναι ακριβείς και σχετικές με τα σενάρια του πραγματικού κόσμου. Η μελέτη αξιολογεί την απόδοση της MySQL υπό δύο διακριτές συνθήκες: με και χωρίς indexing.

Οι δοκιμές που πραγματοποιήθηκαν στη MySQL είναι λεπτομερείς, με έμφαση στη μέτρηση των ρυθμών data ingestion, των χρόνων απόκρισης ερωτημάτων και της συνολικής αποτελεσματικότητας του συστήματος σε διαφορετικά λειτουργικά σενάρια.

Ένα από τα βασικά ευρήματα είναι η μεταβολή της απόδοσης που παρατηρείται όταν η MySQL λειτουργεί με και χωρίς indexing. Η μελέτη ποσοτικοποιεί αυτή τη διαφορά, παρέχοντας στατιστικά δεδομένα για την ταχύτητα και την αποτελεσματικότητα επεξεργασίας

Η βασική ανάλυση MySQL θέτει το στάδιο για μια μεταγενέστερη συγκριτική μελέτη με τα TSDB.

### Οι μετρήσεις της InfluxDB και Apache Druid

Εισαγωγή στις μετρήσεις:

Η ενότητα "Μετρήσεις για InfluxDB και Apache Druid" παρουσιάζει μια λεπτομερή εμπειρική ανάλυση που συγκρίνει την απόδοση του InfluxDB και του Apache Druid στον χειρισμό δεδομένων χρονοσειρών μεγάλης κλίμακας.

Η μεθοδολογία περιλαμβάνει μια σειρά δοκιμών που έχουν σχεδιαστεί για την αξιολόγηση και των δύο βάσεων δεδομένων υπό διάφορες συνθήκες. Αυτές οι συνθήκες αναπαράγουν τυπικά λειτουργικά σενάρια, συμπεριλαμβανομένων βάσεων δεδομένων σε διαφορετικά στάδια πλήρωσης (κενό, μερικώς γεμάτο, πλήρως φορτωμένο) και με ποικίλες δομές δεδομένων. Οι δοκιμές διεξάγονται σε ελεγχόμενο περιβάλλον για να διασφαλιστεί η ακρίβεια και η αξιοπιστία των αποτελεσμάτων.

Περιγράφεται λεπτομερώς η απόδοση των TSDB σε διάφορα σενάρια. Βασικές μετρήσεις, όπως τα ποσοστά απορρόφησης δεδομένων, οι χρόνοι απόκρισης ερωτημάτων και η αποτελεσματικότητα του συστήματος παρακολουθούνται στενά. Η απόδοση του InfluxDB σημειώνεται ότι είναι ιδιαίτερα ισχυρή σε σενάρια που περιλαμβάνουν επεξεργασία δεδομένων σε πραγματικό χρόνο και εγγραφή δεδομένων μεγάλου όγκου, επιδεικνύοντας την ικανότητά του να χειρίζεται αποτελεσματικά μεγάλα σύνολα δεδομένων. Ομοίως, το Apache Druid επιδεικνύει τα δυνατά του σημεία σε σενάρια που απαιτούν ανάλυση σε πραγματικό χρόνο και αποθήκευση ιστορικών δεδομένων, ευθυγραμμισμένα με το σχεδιασμό και την αρχιτεκτονική του.

## Στρατηγικές Αναβάθμισης Υπαρχόντων Συστημάτων με Ενσωμάτωση TSDB

Το κεφάλαιο αυτό εστιάζει στην ενσωμάτωση Βάσεων Δεδομένων Χρονοσειρών (TSDB) σε κατανεμημένες αρχιτεκτονικές,. Περιστρέφεται από μια συζήτηση των πλεονεκτημάτων απόδοσης των TSDB σε σχέση με τις παραδοσιακές βάσεις δεδομένων MySQL σε πρακτικές στρατηγικές για την αναβάθμιση των υπαρχόντων συστημάτων για την ενσωμάτωση των TSDB. Αυτό περιλαμβάνει δύο κύριες οδούς: την αντικατάσταση της υπάρχουσας αποθήκης δεδομένων με ένα TSDB ή την υιοθέτηση μιας πιο ολοκληρωμένης προσέγγισης αντικαθιστώντας τόσο την αποθήκη δεδομένων όσο και την υπάρχουσα σχεσιακή βάση δεδομένων με μια ενοποιημένη λύση TSDB.

Η αρχιτεκτονική του παλαιού συστήματος, σχεδιασμένη για τη διαχείριση δεδομένων της αγοράς ενέργειας, αποτελείται από πολλά στοιχεία: μια διεπαφή εφαρμογής front-end για αλληλεπίδραση με τον χρήστη, ένα Energy Markets Data API για πρόσβαση σε δεδομένα και διάφορα στοιχεία διαχείρισης δεδομένων, όπως ένα ORM για αλληλεπίδραση με βάση δεδομένων και MySQL για αποθήκευση δεδομένων. Το σύστημα περιλαμβάνει επίσης API με προβολές SQL για δυναμική αναζήτηση και ένα στοιχείο αποθήκευσης δεδομένων που αντιμετωπίζει προκλήσεις όσον αφορά την πολυπλοκότητα, την επιρρεπή σε σφάλματα και τις ποινές απόδοσης, ιδιαίτερα σε εργασίες συγκέντρωσης δεδομένων.

Η προτεινόμενη αναβάθμιση περιλαμβάνει την ενσωμάτωση ενός TSDB στην αρχιτεκτονική, βελτιώνοντας τις δυνατότητες επεξεργασίας και συγκέντρωσης δεδομένων. Αυτή η αλλαγή περιλαμβάνει τη διατήρηση ορισμένων υπαρχόντων στοιχείων, όπως το ORM Data Access και το MySQL RDBMS, για σταθερότητα, ενώ εισάγεται ένα νέο επίπεδο αντιστοίχισης RDB και μια βάση δεδομένων χρονοσειρών για αποτελεσματικό χειρισμό δεδομένων και προηγμένες αναλύσεις. Το σύστημα διατηρεί τον αρθρωτό σχεδιασμό του, εξασφαλίζοντας επεκτασιμότητα και συντηρησιμότητα.

Στην τελευταία αρχιτεκτονική επανάληψη, το σύστημα υφίσταται περαιτέρω βελτιώσεις με την πλήρη ενσωμάτωση ενός TSDB, αντικαθιστώντας την παραδοσιακή σχεσιακή βάση δεδομένων. Αυτή η μετατόπιση βελτιώνει τους χρόνους απόκρισης ερωτημάτων και μειώνει την αντιγραφή δεδομένων, αλλά απαιτεί σημαντική επανεγγραφή του επιπέδου API. Παρά τις αλλαγές στο backend, η αλληλεπίδραση του frontend παραμένει συνεπής, διασφαλίζοντας μια σταθερή εμπειρία χρήστη και επιδεικνύοντας την προσαρμοστικότητα και τη δέσμευση του συστήματος να εξισορροπεί την καινοτομία με τη λειτουργική σταθερότητα. Αυτή η εξέλιξη αντικατοπτρίζει μια στρατηγική προσέγγιση για τη βελτίωση των δυνατοτήτων χειρισμού δεδομένων του συστήματος με έμφαση στα δεδομένα χρονοσειρών, διασφαλίζοντας παράλληλα τη συνέχεια και την αξιοπιστία στις αλληλεπιδράσεις των χρηστών και τις λειτουργίες του συστήματος.

## Επισκόπηση εργασίας:

Τέλος η  εργασία έχει αναλύσει τη σύνθετη δυναμική της απόδοσης, της επεκτασιμότητας και της λειτουργικότητας της βάσης δεδομένων στο πλαίσιο της διαχείρισης δεδομένων μεγάλης κλίμακας, ευαίσθητων στο χρόνο.

Τα εμπειρικά δεδομένα που ελήφθησαν από διάφορες δοκιμές έδειξαν ότι το InfluxDB και το Apache Druid, ως TSDB, γενικά υπερτερούν της MySQL στον χειρισμό δεδομένων χρονοσειρών. Αυτή η υπεροχή αποδίδεται στην αρχιτεκτονική και το σχεδιασμό τους, τα οποία είναι ειδικά βελτιστοποιημένα για δεδομένα με χρονική σήμανση.

Τονίσαμε επίσης τις προκλήσεις όσον αφορά τον μετασχηματισμό του σχήματος δεδομένων, την προσαρμογή της γλώσσας ερωτημάτων και τη συνολική μετεγκατάσταση συστήματος. Η γνώση αυτή είναι χρήσιμη για τους οργανισμούς,  που εξετάζουν το ενδεχόμενο στροφής σε TSDB για τις ανάγκες διαχείρισης δεδομένων τους. Μία από τις βασικές συστάσεις της εργασίας είναι η υιοθέτηση μιας υβριδικής προσέγγισης,  για την αξιοποίηση των εξειδικευμένων δυνατοτήτων των TSDB για λειτουργίες δεδομένων που βασίζονται στον χρόνο, διατηρώντας παράλληλα την ευελιξία και την ευρωστία της MySQL για τη γενική διαχείριση δεδομένων.

Δεδομένου του αυξανόμενου όγκου και της πολυπλοκότητας των δεδομένων σε διάφορους τομείς, τα TSDB όπως το InfluxDB και το Apache Druid είναι έτοιμες να γίνουν αναπόσπαστα στοιχεία της σύγχρονης υποδομής δεδομένων. Οι τελικές παρατηρήσεις τονίζουν την ανάγκη για μια προσεκτική και καλά σχεδιασμένη προσέγγιση κατά την ενσωμάτωση των TSDB στα υπάρχοντα συστήματα διαχείρισης δεδομένων. Η εργασία συνιστά στους οργανισμούς να λαμβάνουν υπόψη τις συγκεκριμένες απαιτήσεις δεδομένων και τα λειτουργικά τους πλαίσια όταν αποφασίζουν για την κατάλληλη τεχνολογία βάσης δεδομένων.

# 6. Contents

# 1. Introduction

## Timeseries Databases, Timeseries Data and the Difference form Other Types of Databases

A Time Series Database (TSDB) is a specialized form of database engineered for the optimal handling of time-stamped or time series data. Time series data comprises measurements or events systematically recorded, monitored, downsampled, and aggregated over temporal intervals. This encompasses a wide range of analytical data types, such as server metrics, application performance metrics, network data, sensor data, event logging, user interactions such as clicks, and financial transactions in markets. The architectural foundation of a TSDB is intrinsically designed to manage metrics and measurements with time stamps effectively. TSDBs are particularly fine-tuned to facilitate the analysis of temporal change.

Unique characteristics of time series data that distinguish it from other data workloads include sophisticated data lifecycle management, data summarization techniques, and the capacity for extensive range scans across numerous records.[10] Time series data is an aggregation of observations gleaned through repeated measurements over designated time periods. When visualized, these data points form a graph where one axis invariably represents time. In the context of metrics, time series data can be understood as specific data points tracked sequentially over time intervals. For example, a metric might track the daily variation in inventory sales in a retail environment. [9] Given that time is a fundamental element of all observable phenomena, time series data is ubiquitous. The proliferation of instrumentation in contemporary society means that sensors and systems are continuously generating a vast stream of time series data. This data has many applications across various sectors. To illustrate, time series analysis is employed in diverse fields, ranging from monitoring electrical activity in the brain, managing server logs, tracking stock market prices, analyzing annual retail sales trends, to measuring heart rate.

Time series databases are underpinned by key architectural designs that set them apart from traditional databases. These include optimized storage and compression of time-stamped data, comprehensive data lifecycle management, effective data summarization, the ability to conduct extensive scans of time series dependent records, and the execution of time series aware queries. For instance, time series databases are adept at summarizing data over extended periods. This involves analyzing a range of data points to compute metrics such as the percentile increase of a specific measure over a given period, compared to the same period in the previous year, with the summary organized monthly. Such computational tasks are challenging to optimize in distributed key-value stores. However, TSDBs are specifically optimized for these scenarios, delivering millisecond-level query response times for data spanning several months. Another example of TSDB functionality is the management of high-precision data for short durations. This data is subsequently aggregated and downsampled into longer-term trend data. In a TSDB, data points are systematically deleted after their relevance period expires. Implementing this type of data lifecycle management is challenging in standard databases, requiring developers to devise efficient schemes for the large-scale eviction and summarization of data. TSDBs inherently provide these capabilities, simplifying the management of time series data for application developers.

## Literature Review

The increasing volume and complexity of time-stamped data in various sectors, particularly with the advent of the Internet of Things (IoT) and smart devices, have led to a significant interest in the efficient management of time series data. This literature review explores various approaches and technologies for handling time series data, with a focus on Time Series Database Management Systems (TSDBMS) such as InfluxDB, and their comparison with traditional SQL databases.

In their work "Hybrid Approach for Efficient Data Management" Leighton et al. (2023) [16] discuss a hybrid system combining a Python-based proxy, InfluxDB, and a Sensor Observation Service (SOS) implementation for managing environmental data with complex metadata. They emphasize the challenges of delivering large volumes of time series data efficiently and propose a solution that combines the high performance of TSDBMS with the rich metadata capabilities of SOS. The performance tests show a significant improvement over a standalone SOS setup, demonstrating the potential of hybrid systems in scientific domains where rapid data retrieval and analysis are critical.

In "Comparative Analysis of TSDBMS", Rudakov et al. (2023) [17] provide a comprehensive comparison of four major TSDBMS: InfluxDB, MongoDB, TimescaleDB,[8] and ClickHouse. Their study evaluates these databases based on criteria such as writing and reading speed, database size, and API convenience. The findings suggest that ClickHouse, despite some limitations, stands out for its exceptional speed in both writing and reading operations, making it suitable for storing time series data in industrial applications.

In the same context, Praschl et al. (2022) in "Performance of Specialized Time Series Databases" [18] evaluate the performance of various DBMSs, including InfluxDB, MongoDB, PostgreSQL, TimescaleDB, and LeanXcale, in the context of storing time series data. They conclude that InfluxDB, a specialized time series NoSQL DBMS, shows the best performance in terms of write-throughput and resource utilization. The study underscores the efficiency of InfluxDB in a single server environment, suggesting its suitability for managing high-frequency, time-related data.

Verner-Carlsson and Lomanto (2023) [19] compare MongoDB and PostgreSQL with the Timescale extension for handling time series data, focusing on query execution time. Their study reveals that the performance of these DBMSs varies depending on the query's nature. While MongoDB performs well for queries over a large timespan, TimescaleDB is efficient for queries involving restricted periods or large data volumes. The study highlights the importance of considering the specific requirements of time series data and queries when selecting a DBMS.

## Integration with legacy systems using RDBMSs.

This thesis aims to build upon the findings of these studies by exploring the integration of TSDB like InfluxDB and Apache Druid [4] with traditional relational databases, particularly MySQL. The goal is to assess their comparative performance and identify the best practices for managing large-scale time series data, especially in the energy sector. The motivation for this approach stems from the fact that many existing software systems do time series management using traditional relational databases in less-than-efficient ways. However, due to the size, complexity and costs involved, it is not usually possible to migrate such systems to TSDBMSs. We therefore examine performance and alternative architectural options for even partial migrations of legacy systems.

The reviewed literature suggests a growing trend towards specialized TSDBMS for handling time-stamped data due to their superior performance and resource efficiency. However, the database system choice depends on specific use cases and the data's nature. This thesis contributes to this field by providing insights into the integration of TSDBMS with traditional SQL databases, addressing the challenges and opportunities in this evolving landscape.

# 2. Time series Databases Selection and Comparison

In our comparative analysis, the initial database selected was InfluxDB, as it consistently appears at the top of lists in both DB-Engines [1] and G2 [13], indicating its prominence in the field. Subsequently, Apache Druid was chosen as the second database for evaluation. The decision to include Apache Druid was influenced by its association with the Apache Foundation, a renowned entity in the software development community, which positions it as a natural starting point for this kind of comparative study. The comparison between InfluxDB and Apache Druid is a pertinent topic in the realm of database technologies, particularly when dealing with time series and online analytical processing (OLAP) workloads. This discussion centers around the architectural differences, use cases, and scalability features of both databases, providing insights into their respective strengths and weaknesses.[12]

**InfluxDB** is a time series database with a cloud-native architecture that can operate as a managed cloud service or be self-managed on local hardware. Developed by InfluxData and released in 2013, it is an open-source database designed for high-performance handling of time series data. The latest version, InfluxDB 3.0, built in Rust, offers a decoupled architecture for independent scaling of compute and storage, and supports both SQL and InfluxQL for querying. Its use cases span monitoring [3], IoT, and real-time analytics. Key aspects of InfluxDB's architecture include columnar storage using Parquet and Apache Arrow, a flexible data model with schemaless design, and integrations through HTTP API or the InfluxDB CLI. It also features a decoupled architecture for independent scaling and supports retention policies for data management. Use Cases for InfluxDB primarily include monitoring and alerting, IoT data storage and analysis, and real-time analytics.

**Apache Druid**, on the other hand, is an open-source columnar database tailored for real-time analytics. It emerged from Metamarkets in 2011 and was later contributed to the Apache Software Foundation in 2018. Druid's architecture, a blend of time series databases, search systems, and columnar storage, is optimized for event-driven data and interactive analytics. It is horizontally scalable and supports distributed architectures.[7] Apache Druid's architecture comprises different node types, including Historical, Broker, Coordinator, and MiddleManager/Overlord, each playing a specific role in data management and querying. It uses deep storage for persistent data storage and supports various metadata storage databases. Apache Druid is well-suited for geospatial analysis, machine learning and AI preprocessing, and real-time analytics.

In the context of time series data, both databases have distinct advantages. InfluxDB is specifically designed for such data, offering high write and query performance, making it ideal for applications involving continuous data writing and immediate querying. Apache Druid, while also catering to time series data, emphasizes real-time analytics and the capacity to store historical data in cost-effective storage solutions.

In terms of pricing models, InfluxDB offers a free open-source version, a cloud-based service, and an enterprise edition for on-premises deployment. Apache Druid, being open source, incurs costs related to self-hosting but is also available as a managed service with varying pricing based on service tiers and data management needs.

## The Druid Database

Here we will focus on the deployment of Apache Druid on a single-server architecture. While Apache Druid is often deployed in clustered environments for scalability, this research narrows its scope to a single-server deployment, offering a methodology suitable for environments with limited resources.

The server Prerequisites for this deployment are modest, with a minimum requirement of 6 GiB of RAM. The operating system should be Linux, Mac OS X, or another Unix-like OS, as Windows is currently not supported. Java (11, or 17), Python 3. Prior to installation, a review of Apache Druid's security overview is advised. It is recommended to avoid running Druid under the root user for security reasons. Instead, a dedicated user account should be created for running Druid services.

The Apache Druid 27.0.0 package is downloadable from the official Apache Druid repository. For initiating Druid services, the automatic single-machine configuration is employed. Druid is configured to utilize up to 80% of the available system memory. However, this can be explicitly set by passing a value to the memory parameter.

### The Druid Architecture

In the deployment architecture of Druid, a distributed system, the configuration is delineated across a cluster comprising one or more servers functioning through multiple processes. The deployment encompasses three distinct types of server nodes:

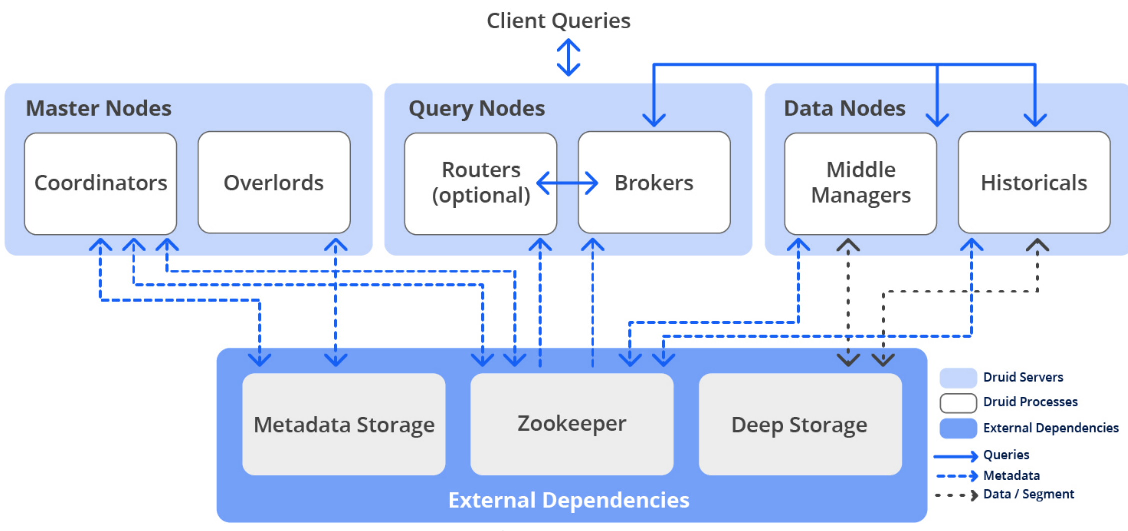


Figure 1. Architecture Diagram of a Distributed Data Storage and Query System

**Master Nodes:** These nodes are pivotal in orchestrating data accessibility and overseeing the ingestion process.
**Query Nodes:** Responsible for receiving and processing queries, these nodes execute the queries across the Druid system and return the resultant data.

**Data Nodes:** These nodes are tasked with the execution of ingestion workloads and the storage of queryable data.

In configurations of a more modest scale, it is feasible to operate all these nodes collectively on a single server. Conversely, in more expansive deployments, each node type is typically allocated one or more dedicated servers. [14]

Additionally, Druid's operational framework includes three critical external dependencies:

- **Deep Storage:** Utilized as a secondary repository for each data segment, deep storage leverages cloud storage solutions or HDFS (Hadoop Distributed File System). Its primary function is to facilitate the transfer of data among Druid processes and to serve as a resilient data source for system recovery post failures.
- **Metadata Storage:** Constituted by a small-scale relational database system such as Apache Derby, MySQL, or PostgreSQL. This component is essential for data management, storing information pertinent to storage segments, ongoing tasks, and other configurational data.
- **Apache ZooKeeper:** Primarily engaged for service discovery and leader election processes. Although historically employed for a broader range of functions, these have since been transitioned to other mechanisms.

Each node type within the Druid architecture can be scaled independently, allowing for the addition or removal of nodes without inducing system downtime. This flexibility is critical for adapting to varying demands in storage, querying capabilities, or coordination. Druid's inherent design automatically redistributes resources, ensuring that any added capacity is immediately operational.

Druid's architecture is marked by a distributed design that efficiently partitions tables into segments. This design not only balances these segments across servers but also swiftly identifies the segments relevant to a specific query. Subsequently, it maximizes computational efficiency by delegating extensive computational tasks to individual data nodes.

Moreover, Druid incorporates approximation methodologies in its operations. While it can execute exact computations, the system often opts for approximations in processes like ranking, histogram calculations, set operations, and count-distinct computations to enhance speed and efficiency.

## Using Apache Druid

Apache Druid features a dynamic data model that contrasts with the rigid schemas found in traditional relational databases. During the data ingestion phase, Druid ingests data in formats like JSON, CSV, or Parquet, allowing you to specify a schema that outlines dimensions for filtering or grouping, metrics for quantitative analysis, and a timestamp column. This ingestion-time schema definition introduces a flexible approach where different data sources within Druid can have individualized schemas, accommodating a diverse range of data structures.

Druid operates on a schema-on-read basis, applying the schema at query time rather than at the time of data ingestion. This method offers adaptability and is capable of handling changes in the data over time. Data in Druid is stored in segments with accompanying metadata detailing the schema for the data they contain. Additionally, Druid's ingestion process supports rollup, a summarization technique that pre-aggregates data, further influencing the schema by defining how data is aggregated. The combination of these features enables Druid to provide real-time analytics across varied data formats and structures with efficiency and flexibility.

**Ingesting**

To ingest data in druid you can either use streaming or batch ingesting, below we will focus only on batch ingestion.

Batch ingestion in Apache Druid is a process suitable for loading large volumes of data not required to be ingested in real-time. This method is often used for historical data loads or for data sources that do not continuously generate data. Here is how you can perform batch ingestion in Apache Druid. Bellow you can see the typical steps for batch ingestion:

1. **Prepare Your Data:** Make sure your data is in a format that Druid can read and that is accessible to Druid.

2. **Define an Ingestion Spec**: Create a JSON ingestion spec file that tells Druid how to read and ingest the data. This spec includes:

   - `` `dataSchema` ``: Defines the dataSource name, the timestamp column, dimensions, and metrics.

   - `` `ioConfig` ``: Details about where the data is located (like type of input source, file paths, etc.) and the input format (CSV, Parquet, JSON, etc.).

   - `` `tuningConfig` ``: Parameters to control the performance aspects of the ingestion job (like memory settings, max rows per segment, etc.).

   Example template for a batch ingestion spec:

```
{
    "type": "index_parallel",
    "spec": {
      "dataSchema": {
        "dataSource": "your-data-source-name",
        "timestampSpec": {
          "column": "timestamp",
          "format": "auto"
        },
        "dimensionsSpec": {
          "dimensions": [
            // List of dimensions
          ]
        },
        "metricsSpec": [
          // List of metrics
        ]
      },
      "ioConfig": {
        "type": "index_parallel",
        "inputSource": {
          "type": "local",
          "baseDir": "path/to/data",
          "filter": "your-data-file.*"
        },
        "inputFormat": {
          // Specify format (json, csv, etc.)
        }
      },
      "tuningConfig": {
        // Tuning parameters
      }
    }
  }
```

Code Snippet 1. Ingestion Spec for Apache Druid

3. **Submit the Task**: Use Druid's console or REST API to submit the ingestion task. This can be done with a command like the one used for streaming ingestion:

```
curl -X POST -H 'Content-Type: application/json' -d @ingestion-spec.json
http://OVERLORD_HOST:PORT/druid/indexer/v1/task
```

4. **Monitor the Task**: Monitor the status of the ingestion task in the Druid console or via the API. Batch tasks can take a while to complete, especially for large datasets.

5. **Querying Data**: After the batch ingestion task is completed successfully, the data will be available for querying in Druid.

When working with Apache Druid, it's important to ensure that your data conforms to Druid's supported file formats for smooth ingestion. Performance tuning is a crucial step, particularly for substantial datasets or when working with limited resources; it requires adjusting the ingestion tasks for the best possible performance. The data schema, defined during the ingestion specification, must accurately reflect your data's structure to avoid inconsistencies. For handling large datasets, data partitioning becomes vital. Efficient partitioning can significantly enhance query performance in Druid. While Druid excels at real-time data analysis, batch ingestion is an effective method for instances where immediate data visibility isn't a necessity, allowing for the periodic processing of large data volumes for subsequent analytics and reporting tasks.

**Querying**

In the realm of data querying using Apache Druid, practitioners have two primary methods at their disposal: SQL-based queries and Druid Native queries. The flexibility of SQL, a widely familiar querying language, is a significant advantage in this context. Queries formulated in SQL are converted into Druid Native format for execution. This translation process allows users to leverage their existing knowledge and pre-written SQL queries, facilitating a smoother integration with Druid's data processing capabilities.

To illustrate, consider the following example: an SQL query designed to aggregate and order data from the 'wikipedia' datasource is presented alongside its equivalent Druid Native query. The SQL query:

```sql
SELECT
  "page",
  "countryName",
  COUNT(*) AS "Edits"
FROM "wikipedia"
GROUP BY 1, 2
ORDER BY "Edits" DESC
```

Code Snippet 2. SQL Query example

This is translated into the following Druid Native format:

```
{
    "queryType": "groupBy",
    "dataSource": {
        "type": "table",
        "name": "wikipedia"
    },
    "intervals": {
        "type": "intervals",
        "intervals": [
            "-146136543-09-08T08:23:32.096Z/146140482-04-24T15:36:27.903Z"
        ]
    },
    "granularity": {
        "type": "all"
    },
    "dimensions": [
        {
            "type": "default",
            "dimension": "page",
            "outputName": "d0",
            "outputType": "STRING"
        },
        {
            "type": "default",
            "dimension": "countryName",
            "outputName": "d1",
            "outputType": "STRING"
        }
    ],
    "aggregations": [
        {
            "type": "count",
            "name": "a0"
        }
    ],
    "limitSpec": {
        "type": "default",
        "columns": [
            {
                "dimension": "a0",
                "direction": "descending",
                "dimensionOrder": {
                    "type": "numeric"
                }
            }
```
Code Snippet 1. Analogues Ingestion Spec

Additional examples further demonstrate the translation from SQL to Druid Native queries. For instance, a SQL query retrieving sum aggregates of a specific field within a specified time interval, grouped by multiple dimensions, and ordered descending, is seamlessly converted into its corresponding Druid Native configuration. This process underscores the versatility of using SQL as an entry point for Apache Druid, particularly for those already versed in SQL syntax and logic.

## The InfluxDB Database

In the realm of InfluxDB documentation, a multitude of installation methods are presented, catering to diverse platforms such as Linux, Windows, Docker, Kubernetes, and even Raspberry Pi. However, our focus will be narrowed to the realm of local development on a single Linux server. [18] The outlined procedure focuses on utilizing systemd for service management. This method is recommended for users seeking an automated approach to manage the InfluxDB service. The installation involves:

a. **Downloading the Package**: Obtain the appropriate .deb or .rpm package from the InfluxData downloads page. For instance, for Ubuntu/Debian AMD64, use:

```
curl -O https://dl.influxdata.com/influxdb/releases/influxdb2_2.7.4-1_amd64.deb
```

b. **Installing the Package**: Utilize the distribution's package manager to install InfluxDB. For Debian-based systems, use:

```
      sudo dpkg -i influxdb2_2.7.4-1_amd64.deb
```

c. **Starting the Service**: Activate the InfluxDB service via:

```
      sudo service influxdb start
```

d. **Service Verification**: Confirm the service status with:

```
      sudo service influxdb status
```

Post-installation configuration may be necessary to align InfluxDB with specific system requirements. This includes setting directory permissions and customizing the network port (default: TCP port 8086). InfluxDB sends telemetry data to InfluxData by default. To disable this, start InfluxDB with the *--reporting-disabled* flag. The initial setup process includes creating an organization, a primary bucket, and an admin authorization. Setup can be performed through the InfluxDB UI or CLI.

## The Influx Architecture

InfluxDB 3.0, formerly known as InfluxDB IOx, represents a significant evolution in the realm of scalable databases, particularly in the context of time series data management. Its architecture is characterized by its segmentation into four distinct components, each responsible for a specific operational aspect: data ingestion, data querying, data compaction, and garbage collection. These components function quasi-independently, underpinning the database's performance in both data loading and querying processes. [15]

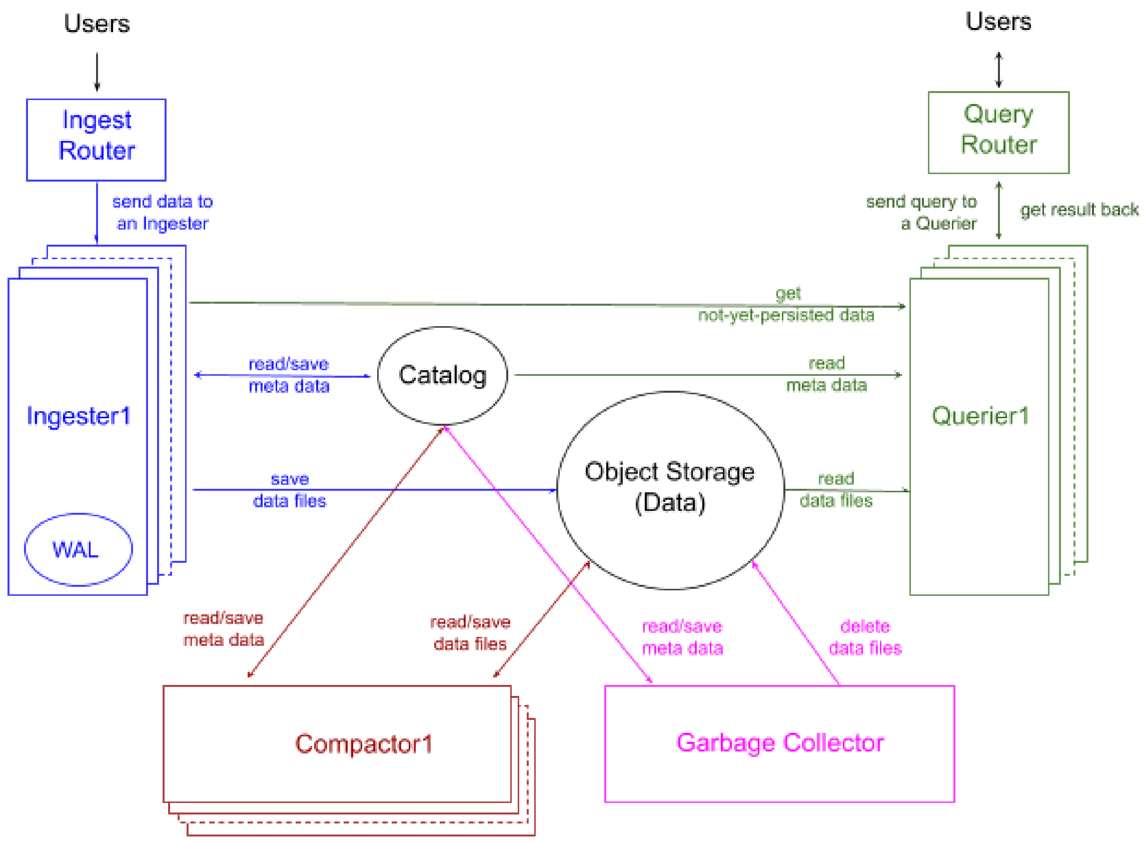


Figure 2. InfluxDB architecture

The data ingestion process is facilitated through a component structure that includes an Ingest Router and multiple Ingesters, allowing dynamic scalability in response to varying data workloads. This component is crucial for the initial handling of incoming data, which involves table identification, schema validation, data partitioning, deduplication, and persistence. Data querying in InfluxDB 3.0 is handled by a Query Router and Queriers. The Queriers are instrumental in executing query plans, which involve metadata caching, data reading and caching, and deduplication processes. This component is optimized to handle high query workloads efficiently.

The data compaction component is designed to address the challenges posed by the creation of numerous small files during the ingestion phase. By compacting these files into larger and non-overlapped files, the database enhances its query performance. InfluxDB 3.0 incorporates a garbage collection mechanism to manage data retention and space reclamation. This process involves the scheduling of background jobs for soft and hard deletion of data.

The architecture of InfluxDB 3.0 includes two primary storage types: Catalog and Object Storage. The Catalog is dedicated to cluster metadata, while Object Storage, which can be integrated with systems like Amazon AWS S3, is used for storing actual data. InfluxDB 3.0's cluster operation is distinguished by its use of dedicated computational resources and the potential for operating on single or multiple Kubernetes clusters. This approach is pivotal in ensuring the isolation of clusters to mitigate "noisy neighbor" issues and enhance reliability.

## Using InfluxDB

InfluxDB, with its diverse range of data ingestion options, stands as a pivotal tool in modern data management and analysis. The database supports several methods including Influx user interface (UI), HTTP API, influx CLI, Telegraf, and client libraries. Understanding these methods, particularly the line protocol, is vital for effective data handling in InfluxDB.

Line protocol is the cornerstone of data writing in InfluxDB. It is a text-based format that structures data points for InfluxDB. The protocol contains essential elements: measurement, tag set, field set, and timestamp. Each element plays a critical role in defining and organizing data in the database.

The elements of Line Protocol include the Measurement, which is a string identifying the data's measurement; the Tag Set, comprising key-value pairs used for indexing and querying; the Field Set, containing key-value pairs that hold the data values; and the Timestamp, a Unix timestamp that provides the time context for the data. Parsing Line Protocol elements involves determining the measurement from the text before the first comma, extracting tag sets from key-value pairs following the measurement, identifying field sets between the first and second whitespace, and recognizing timestamps as the integer values after the second whitespace.

Constructing line protocol involves understanding its syntax and structure. For example, in a home sensor data scenario measuring temperature, humidity, and carbon monoxide, the line protocol is structured as follows:

```
home,room=Living\ Room temp=21.1,hum=35.9,co=0i 1641024000
```

Here, 'home' is the measurement, 'room' is a tag, and 'temp', 'hum', and 'co' are field sets with respective values and a timestamp.

Various methods for writing data in line protocol format to InfluxDB include using the InfluxDB UI for a graphical interface, the influx CLI for command-line interaction, and the InfluxDB API for data writing via HTTP requests.

An example using cURL for the InfluxDB API:

```
curl --request POST
"http://localhost:8086/api/v2/write?org=myOrg&bucket=myBucket&precision=s" \
 --header "Authorization: Token myToken" \
 --data-raw "home,room=Living\ Room temp=21.1,hum=35.9,co=0i 1641024000"
```

InfluxDB offers a range of tools for querying data, crucial for extracting meaningful insights from time series datasets. The document discusses the use of both the Flux language, designed specifically for InfluxDB and other data sources, and InfluxQL, a SQL-like query language tailored for time series data in InfluxDB.

Flux, a functional scripting language, provides flexibility and power in querying and processing data from InfluxDB. Key functions in Flux include from(), for data retrieval, range(), for time-bound data filtering, and filter(), for column value-based data filtering. Flux utilizes a pipe-forward operator (|>) for seamless function chaining and Flux queries can be executed through the InfluxDB UI, influx CLI, or InfluxDB API. A typical Flux query might look like this:

```
from(bucket: "get-started")
  |> range(start: 2022-01-01T08:00:00Z, stop: 2022-01-01T20:00:01Z)
  |> filter(fn: (r) =>  r._measurement == "home")
  |> filter(fn: (r) => r._field == "co" or r._field == "hum" or r._field ==
"temp")
```

On the other hand InfluxQL offers a SQL-like experience for querying time series data from InfluxDB, especially suitable for versions 0.x and 1.x of InfluxDB. Basic InfluxQL queries involve SELECT, FROM, and optional WHERE clauses. An example of an InfluxQL query:

```
SELECT co, hum, temp, room FROM "get-started".autogen.home WHERE time >= '2022-
01-01T08:00:00Z' AND time <= '2022-01-01T20:00:00Z'
```

InfluxQL queries can also be executed via the influx CLI or InfluxDB API. An example using the InfluxDB API:

```
curl --get "http://localhost:8086/query?org=myOrg&bucket=get-started" \
 --header "Authorization: Token myToken" \
 --data-urlencode "q=SELECT co, hum, temp, room FROM home WHERE time >= '2022-01-
01T08:00:00Z' AND time <= '2022-01-01T20:00:00Z'"
```

Data processing in Flux often involves the map() function, which iterates over each data row, allowing for value updates or transformations. An example of the map() function, applied to humidity data:

```
from(bucket: "get-started")
  |> range(start: 2022-01-01T08:00:00Z, stop: 2022-01-01T20:00:01Z)
  |> filter(fn: (r) => r._measurement == "home")
  |> filter(fn: (r) => r._field == "hum")
  |> map(fn: (r) => ({r with _value: r._value / 100.0}))
```

The group() function in Flux is used to regroup data by specific column values, preparing it for subsequent processing steps. An example using the group() function:

```
from(bucket: "get-started")
  |> range(start: 2022-01-01T08:00:00Z, stop: 2022-01-01T20:00:01Z)
  |> filter(fn: (r) => r._measurement == "home")
  |> group(columns: ["room", "_field"])
```

Flux offers aggregate and selector functions to condense or pinpoint specific data from input tables. Using mean() to calculate the average temperature:

```
from(bucket: "get-started")
  |> range(start: 2022-01-01T08:00:00Z, stop: 2022-01-01T20:00:01Z)
  |> filter(fn: (r) => r._measurement == "home")
  |> filter(fn: (r) => r._field == "temp")
  |> mean()
```

For users accustomed to relational databases, Flux provides the pivot() function to transform data into a more familiar relational schema. Pivoting temperature data in a kitchen:

```
from(bucket: "get-started")
  |> range(start: 2022-01-01T14:00:00Z, stop: 2022-01-01T20:00:01Z)
  |> filter(fn: (r) => r._measurement == "home")
  |> filter(fn: (r) => r._field == "temp")
  |> pivot(rowKey: ["_time"], columnKey: ["_field"], valueColumn: "_value")
```

Downsampling, a strategy to reduce the volume of data while preserving trends, is accomplished using functions like aggregateWindow(). Downsampling temperature data over two-hour windows:

```
from(bucket: "get-started")
  |> range(start: 2022-01-01T14:00:00Z, stop: 2022-01-01T20:00:01Z)
  |> filter(fn: (r) => r._measurement == "home")
  |> filter(fn: (r) => r._field == "temp")
  |> aggregateWindow(every: 2h, fn: mean)
```

InfluxDB tasks allow for the scheduling of queries, automating data processing operations. These tasks can perform various operations described earlier and write the results back to InfluxDB.

# 3. Problem statement

In the rapidly evolving world of energy management, handling extensive and complex data efficiently is crucial. This chapter delves into the specific challenges faced when managing large-scale energy data using traditional SQL databases and explores the potential advantages of transitioning to a time series database (TSDB). The focus of our investigation is on a typical dataset of open data for energy markets, available from ENTSO-E via ftp and API, namely "Actual Total Load". This dataset offers the energy (MWh) consumed in European markets. Every month, a new file containing the total load values is offered via ftp. Every day, new lines are added containing data of the previous day for all European energy markets, at various time resolutions (60, 30, or 15 min) and the size of the file grows, until the end of the month where it reaches its maximum size (around 300 MB, 3219201 lines). Important to note is that the number of rows increases close to linearly based on the date of the month because each file has a complete record from the start of the month until the date mentioned on the file. Below you can see said pattern where the number of rows increases until the 30th of April and drops suddenly.

| File Name | Date | Line Count |
|---|---|---|
| 20230425_2023_04_ActualTotalLoad_6.1.A | 2023/04/25 | 95498 |
| 20230426_2023_04_ActualTotalLoad_6.1.A | 2023/04/26 | 113835 |
| 20230427_2023_04_ActualTotalLoad_6.1.A | 2023/04/27 | 118325 |
| 20230428_2023_04_ActualTotalLoad_6.1.A | 2023/04/28 | 122925 |
| 20230429_2023_04_ActualTotalLoad_6.1.A | 2023/04/29 | 135161 |
| 20230430_2023_04_ActualTotalLoad_6.1.A | 2023/04/30 | 139659 |
| 20230502_2023_05_ActualTotalLoad_6.1.A | 2023/05/02 | 3583 |
| 20230503_2023_05_ActualTotalLoad_6.1.A | 2023/05/03 | 9265 |
| 20230504_2023_05_ActualTotalLoad_6.1.A | 2023/05/04 | 13818 |
| 20230505_2023_05_ActualTotalLoad_6.1.A | 2023/05/05 | 18375 |
| 20230506_2023_05_ActualTotalLoad_6.1.A | 2023/05/06 | 22939 |

However, it is common that values may be missing or modified in some next iteration, due to inherent deficiencies of the data collection mechanism of ENTSO-E. This introduces considerable performance challenges for any party collecting such data and maintaining an up-to-date database. Below is the representation of the size of this file for our 49-day window of study, from 24 April 2023 to 13 June 2023.

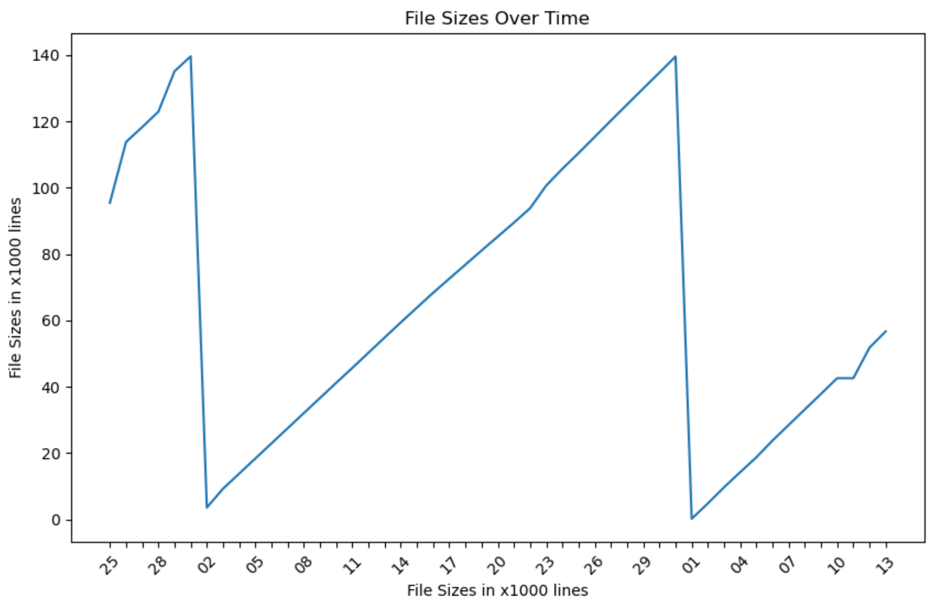


Figure 3: File Size to Date Comparison

## Data Characteristics and Challenges

The dataset under consideration includes entries with various attributes like DateTime, ResolutionCode, AreaCode, and TotalLoadValue. Each entry represents a unique time slot's energy load. The primary challenge in SQL is the labor-intensive process of gap filling and aggregating data. These operations are not only complex but also inefficient in terms of performance. A typical Relational DB schema for the "Actual Total Load" dataset is shown in Figure 1 (source: diem-platform.com). In this section, we shall delve deeper into the intricacies and challenges associated with the dataset in question. Subsequent paragraphs will elucidate through illustrative diagrams the modalities of data transmission from the ENTSO-E system.

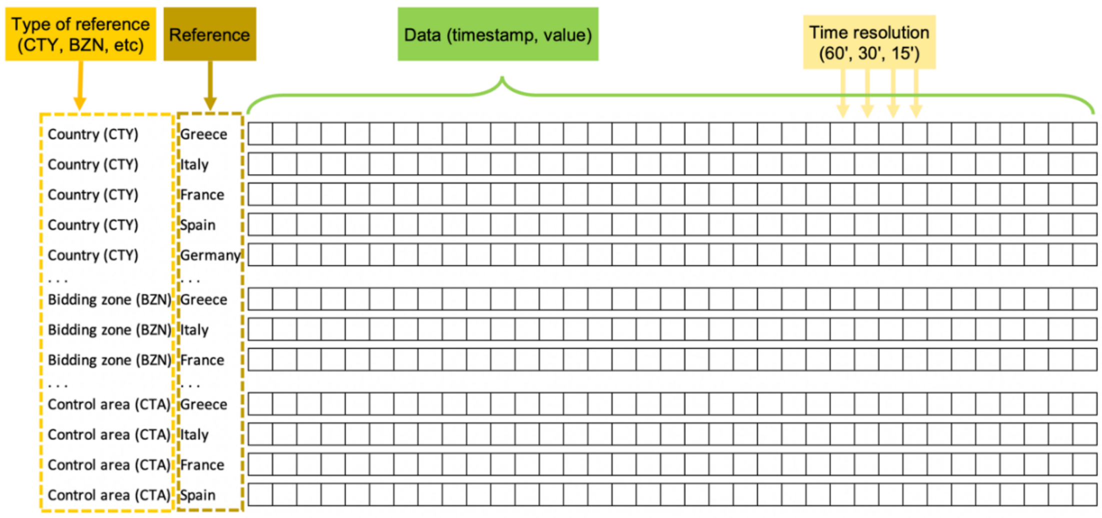


Figure 4. Data series and dimensions for raw data to be imported.

Figure 5. exemplifies the structural composition of the dataset. It commences with a type of reference, succeeded by the reference itself, culminating in a sequential aggregation of data points. Each datum is characterized by a distinct resolution, a timestamp, and a corresponding value. Our methodology entails the acquisition of data batches on a daily cadence, each batch encompassing updated values commencing from the onset of the respective month.

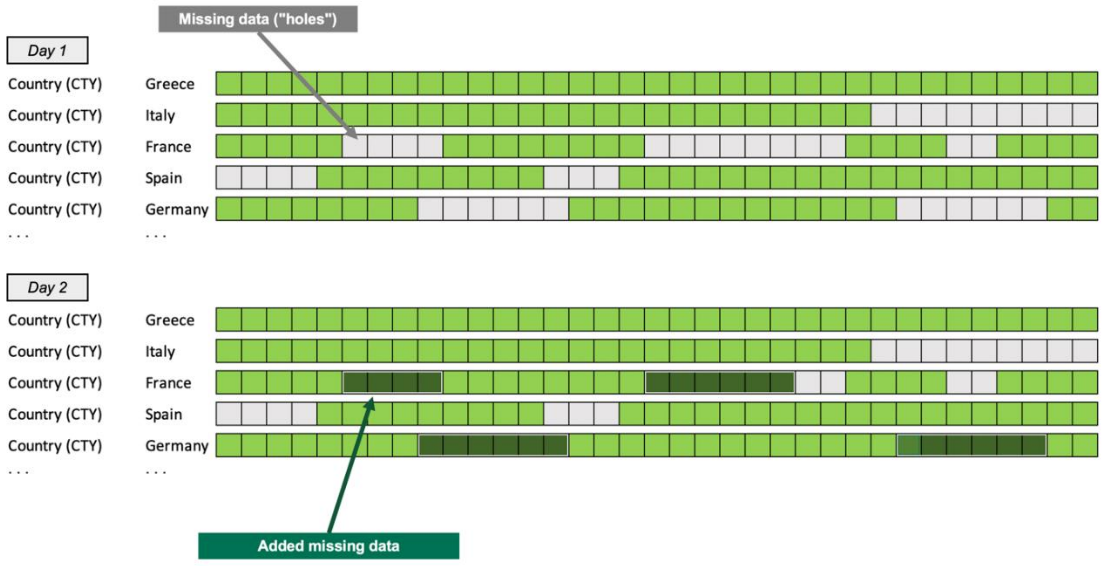


Figure 5. Demonstration of Holes of Missing and Addition to Fil those Holes

It is noteworthy that the initial day's dataset may exhibit lacunae in data representation. However, these gaps are progressively ameliorated in the datasets of the ensuing days. as shown Figure 5

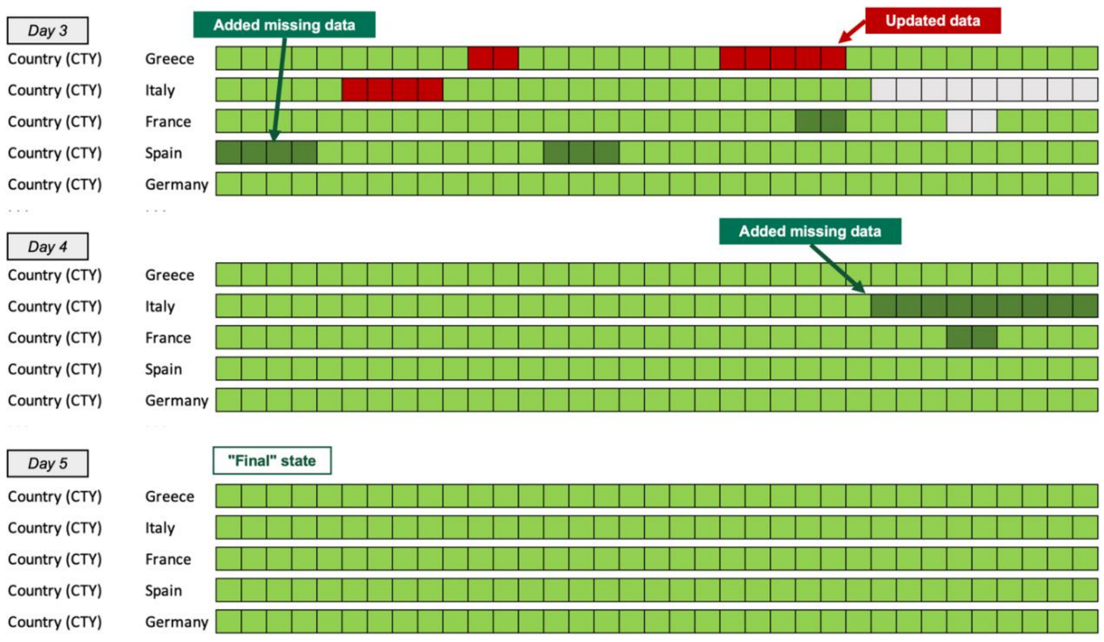


Figure 6. Missing Data Added and Existing Data Updated

Subsequent data batches witness periodic updates in certain values. The dataset, expansive with approximately 3 million lines, intrinsically comprises a mere 100,000 unique additions and 20,000 modifications. This significant disparity in data composition may well account for the observed divergence in performance metrics between the MySQL and TSDB systems. This is shown in Figure 6.

In the following table, we present a detailed breakdown of the daily data transactions, encompassing additions and edits, as extracted from the dataset. This tabular representation is a direct quantitative articulation of the data handling dynamics within the system.

Table 1 the numerical characteristics of the data

| | SIZE AFTER | ADDITIONS | EDITS |
|---|---|---|---|
| *25/4/2023* | 28827 | 28827 | 0 |
| *26/4/2023* | 34325 | 5498 | 1279 |
| *27/4/2023* | 35674 | 1349 | 595 |
| *28/4/2023* | 37061 | 1387 | 675 |
| *29/4/2023* | 40983 | 3922 | 460 |
| *30/4/2023* | 42330 | 1347 | 1070 |
| *2/5/2023* | 1072 | 1072 | 0 |
| *3/5/2023* | 2775 | 1703 | 236 |
| *4/5/2023* | 4141 | 1366 | 442 |
| *5/5/2023* | 5508 | 1367 | 357 |
| *6/5/2023* | 6877 | 1369 | 331 |
| *7/5/2023* | 8246 | 1369 | 361 |
| *8/5/2023* | 9615 | 1369 | 255 |
| *9/5/2023* | 10982 | 1367 | 471 |
| *10/5/2023* | 12349 | 1367 | 434 |

| | SIZE AFTER | ADDITIONS | EDITS |
|---|---|---|---|
| *11/5/2023* | 13700 | 1351 | 545 |
| *12/5/2023* | 15079 | 1379 | 452 |
| *13/5/2023* | 16455 | 1376 | 469 |
| *14/5/2023* | 17817 | 1362 | 812 |
| *15/5/2023* | 19161 | 1344 | 544 |
| *16/5/2023* | 20490 | 1329 | 600 |
| *17/5/2023* | 21762 | 1272 | 364 |
| *18/5/2023* | 23021 | 1259 | 442 |
| *19/5/2023* | 24290 | 1269 | 373 |
| *20/5/2023* | 25537 | 1247 | 452 |
| *21/5/2023* | 26788 | 1251 | 703 |
| *22/5/2023* | 28086 | 1298 | 282 |
| *23/5/2023* | 30223 | 2137 | 432 |
| *24/5/2023* | 31767 | 1544 | 400 |
| *25/5/2023* | 33221 | 1454 | 424 |
| *26/5/2023* | 34695 | 1474 | 1789 |
| *27/5/2023* | 36194 | 1499 | 422 |
| *28/5/2023* | 37669 | 1475 | 915 |
| *29/5/2023* | 39141 | 1472 | 381 |
| *30/5/2023* | 40618 | 1477 | 627 |
| *31/5/2023* | 41685 | 1067 | 655 |
| *1/6/2023* | 64 | 64 | 0 |
| *2/6/2023* | 1460 | 1396 | 3 |
| *3/6/2023* | 2933 | 1473 | 367 |
| *4/6/2023* | 4290 | 1357 | 295 |
| *5/6/2023* | 5608 | 1318 | 253 |
| *6/6/2023* | 7178 | 1570 | 398 |
| *7/6/2023* | 8593 | 1415 | 368 |
| *8/6/2023* | 10006 | 1413 | 353 |
| *9/6/2023* | 11410 | 1404 | 305 |
| *10/6/2023* | 12837 | 1427 | 380 |
| *11/6/2023* | 12837 | 1305 | 340 |
| *12/6/2023* | 15581 | 1439 | 315 |
| *13/6/2023* | 17084 | 1503 | 494 |
| ***TOTAL*** | ***968045*** | ***101099*** | ***22920*** |

The last entry in the table, with a size of 968,045, adds up to 101,099 and edits amounting to 22,920, encapsulates the cumulative transactions throughout the dataset's lifecycle. This cumulative figure underscores the substantial volume of additions and the fewer, yet significant, number of edits. Such a distribution of additions and edits provides an insightful glimpse into the dynamic nature of the dataset and potentially influences the differential performance between MySQL and TSDB systems.

## Experiment Setup and Configurations

The experiment conducted to compare the performance of SQL databases and Time Series Databases (TSDBs) involved a detailed setup across three distinct scenarios. Each scenario was executed on identical servers with 16GB RAM and 4 cores. The first scenario tested empty databases with new data added to an empty table. The second scenario involved databases pre-filled with extensive historical data, comprising 200 tables of 120 months, with new data added to an empty table within this setup. Finally, the third scenario also used databases filled with historical data but added new data to a full table already containing 120 months of data. Performance metrics for these tests included data ingestion response time and query latency.

In the comparative analysis, identical operations were performed in both SQL and the chosen TSDB to assess how each database managed data ingestion and querying under various levels of data volume. A significant focus of the analysis was on the ease of data handling, comparing the need for additional scripting and data transformation in SQL with the more streamlined approach offered by TSDBs. The study aimed to determine if TSDBs could offer at least double the performance efficiency compared to SQL databases. Key aspects of the findings included the broader implications for operational efficiency and potential cost savings, particularly in the energy sector. The study summarized insights into the advantages of using TSDBs for large-scale energy data management and explored the prospects of TSDBs in managing the increasing volume and complexity of data in various industries.

# 4. MySQL Baseline

In our comprehensive study, we embarked on a rigorous evaluation of a MySQL system, carefully configured to mirror a fully loaded database that spans 120 months (about 10 years) of historical data. This meticulous configuration was essential to ensure that our baseline measurements were both accurate and relevant. The primary goal was to evaluate this system's performance under varying conditions, specifically focusing on scenarios with and without indexing. This dual approach was critical in providing a holistic understanding of MySQL's capabilities in different operational settings.

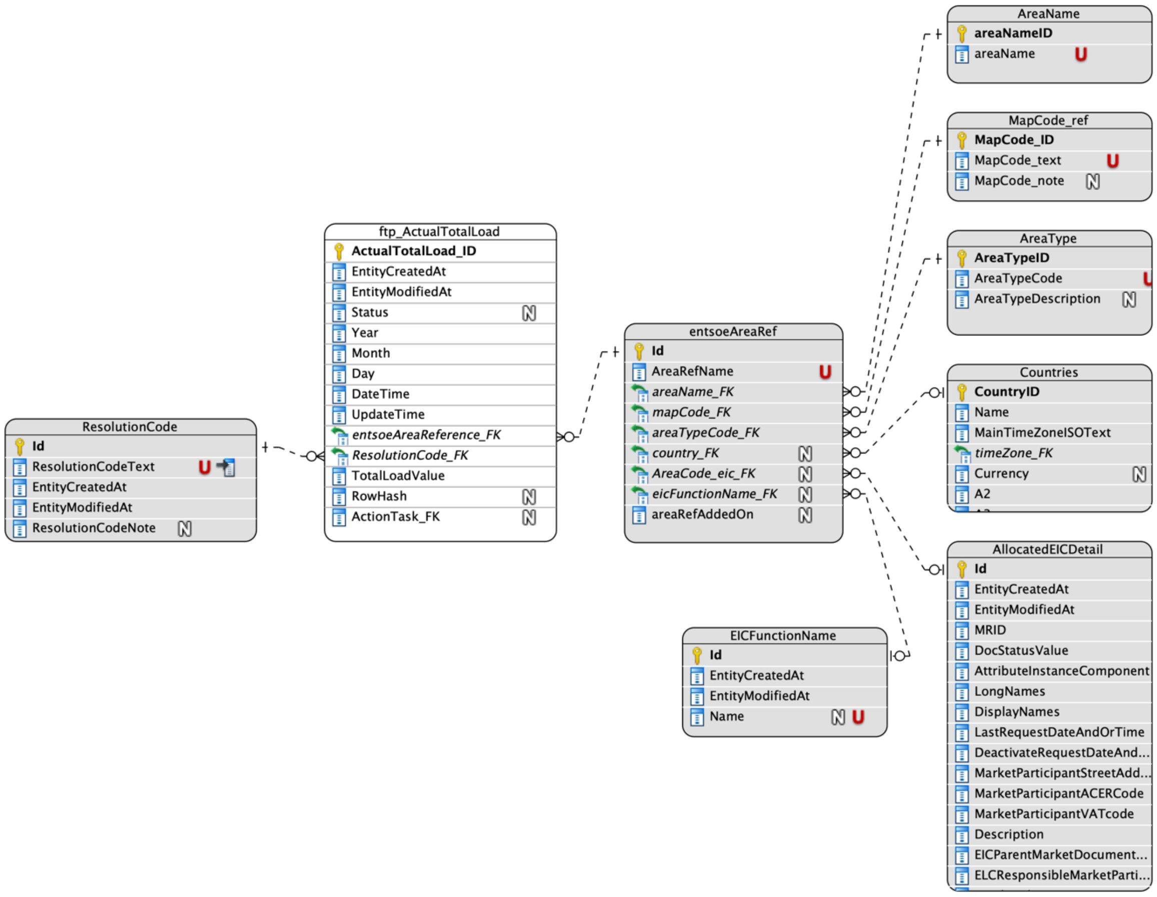


Figure 7 MySQL Database Schema

To ensure a fair and consistent comparative analysis, we loaded the MySQL system with the same dataset to the one used in our Time Series Database (TSBD) assessments. This parity in data characteristics was crucial in eliminating any variables that could potentially bias our results. A particularly interesting finding was an observed correlation between the file size and various performance metrics. This correlation held true across different file sizes, with the notable exception of smaller files. These smaller files appear as "anomalies" in our graph depicting the time consumed to process per

1000 lines; however, this discussion is out of scope of this thesis, as the performance in this case depends on low-level service and systems architectures.

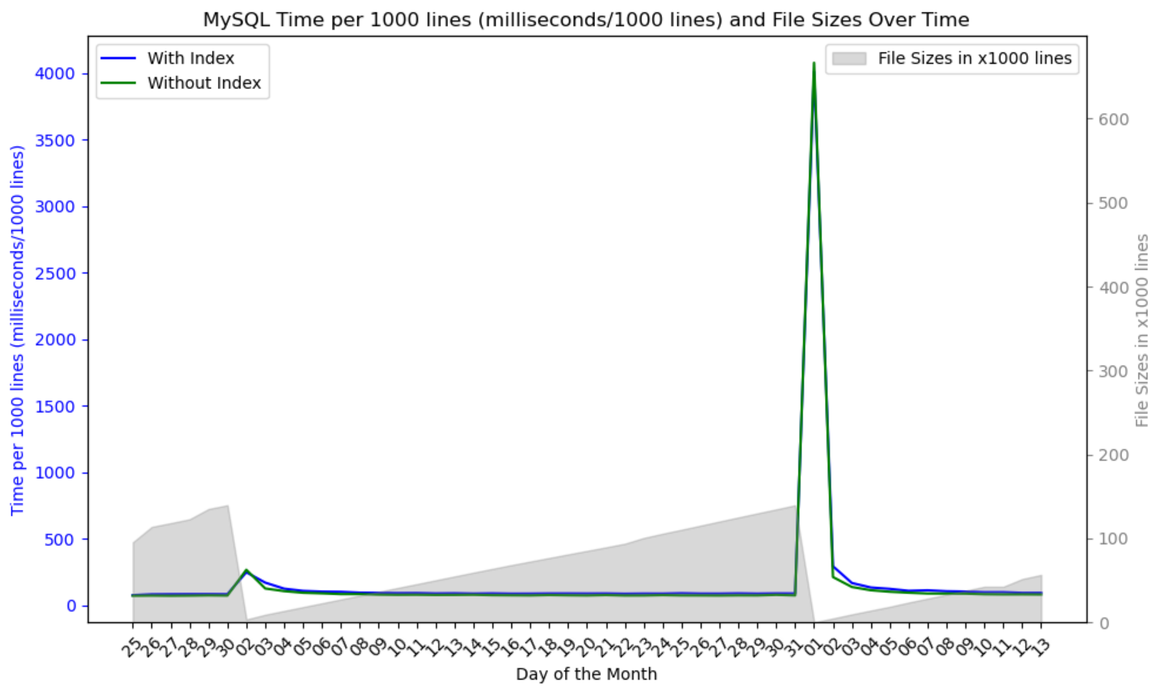


Figure 8 MySQL Time per 1000 Lines and File Sizes Over Time

Moreover, a key highlight of our findings was the average enhancement in processing speed observed when the system operated without indexing. This improvement was quantified at an average of 14.61%, with a standard deviation of 5.23%. This significant increase in speed underscores the impact of indexing on MySQL's performance, which probably stands for other RDBMSs as well.

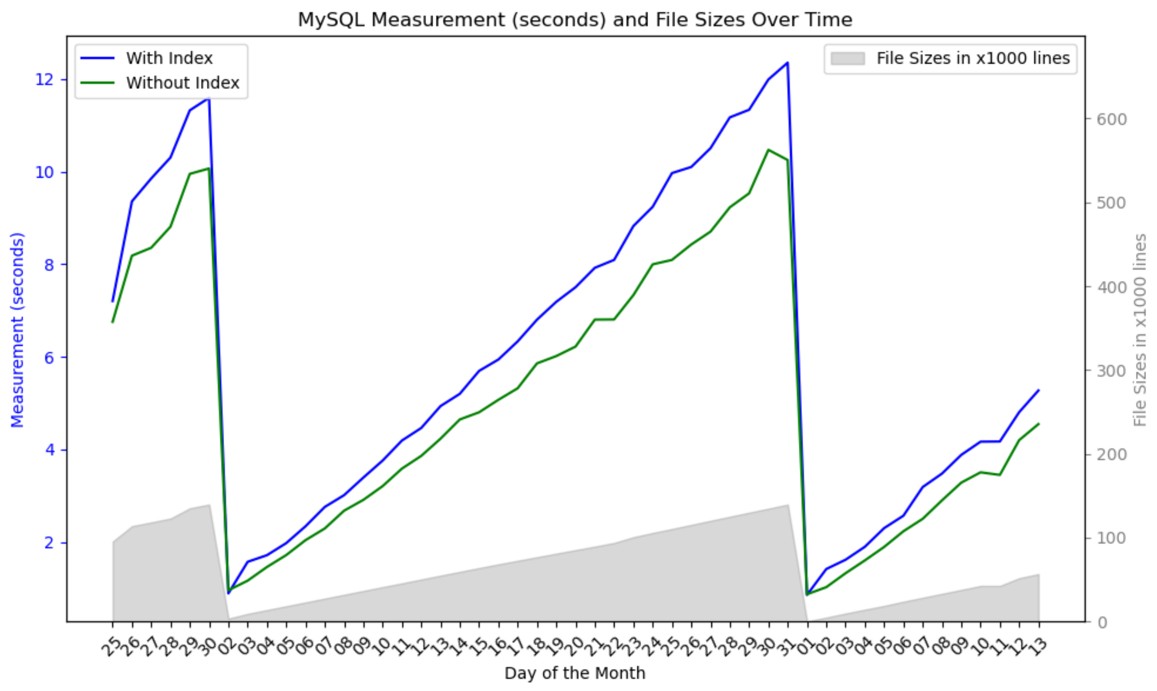


Figure 9 MySQL Measurements and File Sizes Over Time

Below we also have the same data in table form with the measurements in ms.

| Date | FileSize_KB | MySQL_Measurement_Index | MySQL_Measurement |
|---:|---:|---:|---:|
| 25/4 | 9113.6 | 7202.8 | 6753.5 |
| 26/4 | 11264 | 9355.3 | 8181.6 |
| 27/4 | 11264 | 9849.8 | 8353.1 |
| 28/4 | 12288 | 10303.3 | 8808.4 |
| 29/4 | 13312 | 11321.7 | 9949.3 |
| 30/4 | 13312 | 11592.5 | 10066.3 |
| 2/5 | 344 | 886 | 953.4 |
| 3/5 | 880 | 1568.89 | 1161.89 |
| 4/5 | 1331.2 | 1711.9 | 1457.2 |
| 5/5 | 1740.8 | 1972.6 | 1714.5 |
| 6/5 | 2252.8 | 2336.3 | 2036.6 |
| 7/5 | 2662.4 | 2754.6 | 2289.7 |
| 8/5 | 3072 | 3008.7 | 2673.6 |
| 9/5 | 3481.6 | 3389.1 | 2907.6 |
| 10/5 | 3891.2 | 3756.7 | 3204.8 |
| 11/5 | 4403.2 | 4188.9 | 3582.7 |
| 12/5 | 4812.8 | 4460 | 3857.2 |
| 13/5 | 5222.4 | 4932.8 | 4228.7 |
| 14/5 | 5632 | 5197.9 | 4644 |
| 15/5 | 6041.6 | 5693.2 | 4799.1 |
| 16/5 | 6451.2 | 5938.9 | 5068.6 |
| 17/5 | 6860.8 | 6330.8 | 5318 |
| 18/5 | 7270.4 | 6799.1 | 5855.4 |
| 19/5 | 7680 | 7185.4 | 6013.3 |
| 20/5 | 8089.6 | 7500.7 | 6218.5 |
| 21/5 | 8499.2 | 7920.2 | 6801.1 |
| 22/5 | 8908.8 | 8090.9 | 6804.2 |
| 23/5 | 9523.2 | 8822.6 | 7330.5 |
| 24/5 | 10035.2 | 9237.5 | 7995 |
| 25/5 | 11264 | 9965.9 | 8090.8 |
| 26/5 | 11264 | 10098.3 | 8423.5 |
| 27/5 | 12288 | 10504.8 | 8702.3 |
| 28/5 | 12288 | 11170.1 | 9227.5 |

| | | | |
|---|---|---|---|
| 29/5 | 12288 | 11334.1 | 9529.1 |
| 30/5 | 13312 | 11988.5 | 10468.7 |
| 31/5 | 13312 | 12352 | 10248.2 |
| 1/6 | 28 | 853.8 | 868.1 |
| 2/6 | 460 | 1412.9 | 1021.3 |
| 3/6 | 920 | 1613.68 | 1322.6 |
| 4/6 | 1433.6 | 1893.5 | 1599.39 |
| 5/6 | 1843.2 | 2296.5 | 1890.1 |
| 6/6 | 2252.8 | 2565.2 | 2229.4 |
| 7/6 | 2764.8 | 3183.8 | 2497.7 |
| 8/6 | 3174.4 | 3475.1 | 2896.7 |
| 9/6 | 3584 | 3879 | 3279.4 |
| 10/6 | 4096 | 4164.3 | 3502.2 |
| 11/6 | 4096 | 4168.2 | 3446 |
| 12/6 | 4915.2 | 4801.1 | 4196.7 |
| 13/6 | 5427.2 | 5270.7 | 4543.9 |
| **Average** | **6339.820408** | **5924.501429** | **5041.048571** |

# 5. Measurement Methodology and Results for InfluxDB and Apache Druid

Data organization in a non-relational DBMS is always a challenge, especially for those experienced in traditional database design. In our initial attempts using Druid, we adopted a strategy of storing data for each country into separate collections (tables). However, this approach proved to be markedly inefficient. We observed that the time required to process data for a single country was comparable to that needed for incorporating data from all countries into a unified table. This led us to an important realization: the time consumed in filtering data by area name is equivalent to the time taken to add a row to the table. Consequently, we shifted our methodology towards a more consolidated data management approach.

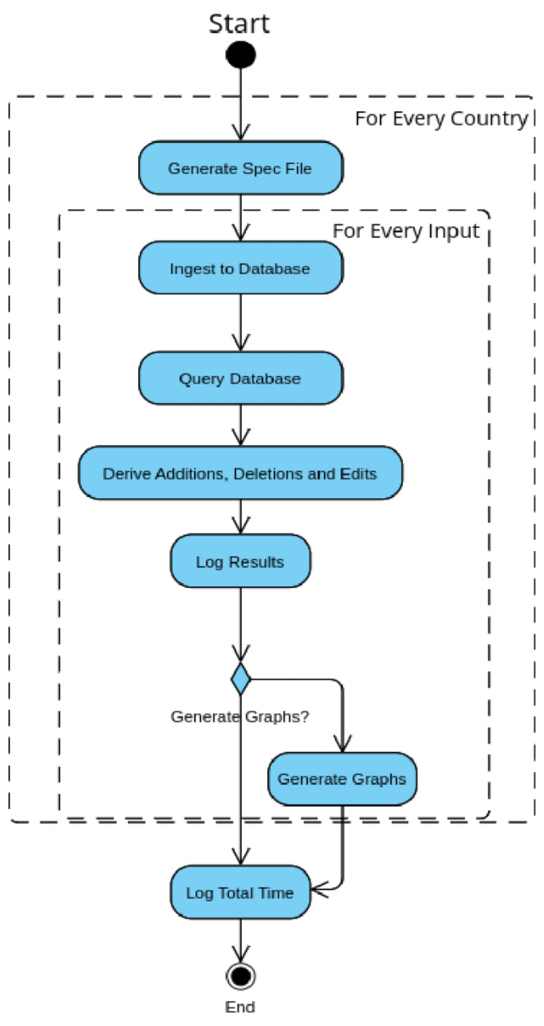


Figure 10 UML Diagram: Data Management via Separate Tables Approach

The structural differences between the two methodologies are shown through our UML diagrams. In the separate tables approach, the primary 'for loop' iterates over each country, creating individual tables, which is then followed by a secondary loop processing the data within each country. This structure, while conceptually straightforward, introduces significant redundancy and processing overhead. Conversely, the consolidated approach reverses these loops. Here, the primary loop processes the data entries, and within this loop, a secondary iteration filters the data by country. This inversion significantly streamlines the data handling process, reducing the computational load, and enhancing efficiency. The reversed loop structure in the consolidated model is not just a syntactical change but represents a fundamental shift in how data is organized and processed, as clearly depicted in the corresponding UML diagram.

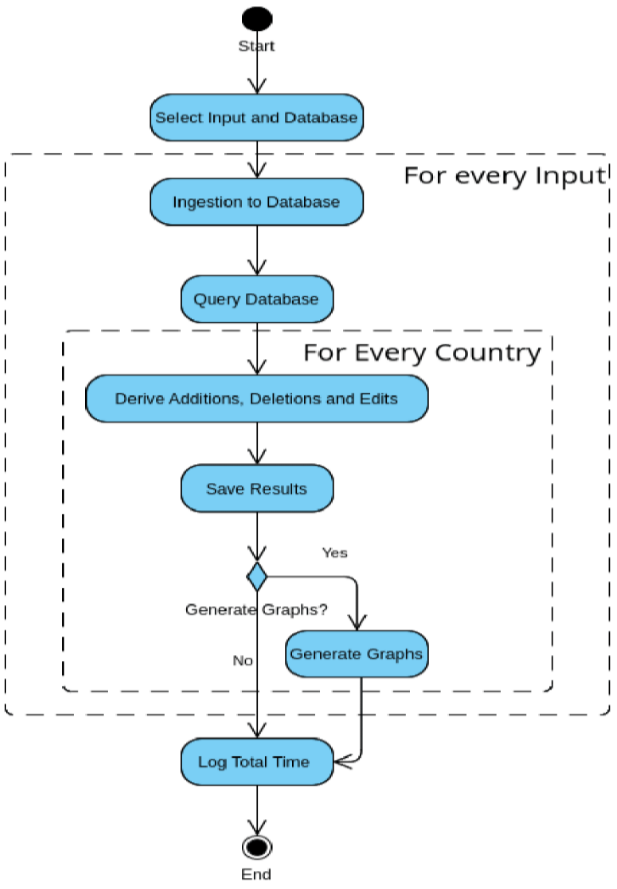


Figure 11. UML activity diagram - consolidated data management flow

## Testing Scenarios

In this section, we explore the system's performance through three scenarios, designed especially for this purpose. These scenarios simulate different states of the database, offering insights into the system's efficiency and scalability. The process, as illustrated in the Activity Diagram, is straightforward, particularly focusing on operations in InfluxDB.

The core of our methodology is a Python script comprising three pivotal functions that manage the data flow from file sources to InfluxDB. The first function, *ingest_and_log_for_all_in_one*, serves as the orchestration hub. It iterates over various file locations, performing a series of operations on each file. These operations include reading and ingesting data from the files into the database, querying the database, calculating data differences across queries, and updating reference tables for the next iteration.

```python
def ingest_and_log_for_all_in_one(input_file_locations, my_measurement_name,
table_before, temp_output_dir):
    for input_file_location in input_file_locations:
        file_in_that_location =
get_dir_location_and_return_the_only_file_in_that_location(input_file_location)
        ingest_duration = ingest_data(file_in_that_location,my_measurement_name)

        table_after, query_duration = query_table(measurement_name)

        derive_addition_deletions_edits(table_after, table_before)

        table_before = copy.deepcopy(table_after)
```

Code Snippet 3. The ingest_and_log_for_all_in_one function.

The second function, ingest_data, is dedicated to ingesting data from a file into InfluxDB. This function performs multiple tasks including reading and cleaning the CSV files by removing unnecessary columns and setting an appropriate index. It then establishes a connection with InfluxDB and writes the data into it. The duration of this operation is measured and returned in milliseconds, providing valuable metrics for performance analysis.

```python
def ingest_data(input_file_location, my_measurement_name=measurement_name):
    df = pd.read_csv(input_file_location, delimiter='\t')
    df.drop(columns=['ResolutionCode', 'AreaCode', 'MapCode', 'UpdateTime'],
inplace=True)
    df.set_index('DateTime', inplace=True)
    with InfluxDBClient(url=url, token=token, org=org) as client:
        with client.write_api(write_options=SYNCHRONOUS) as write_api:
            t1_start = perf_counter()
            write_api.write(bucket, org=org
                            record=df,
                            data_frame_measurement_name=my_measurement_name,
                            data_frame_tag_columns=['AreaName', 'AreaTypeCode'],
                            data_frame_field_columns=['TotalLoadValue'],
                            )
    return int(perf_counter - t1_start)
```

Code Snippet 4. The ingest_data function.

In the third function, query_table, the script queries data from InfluxDB. This function involves constructing and executing a Flux query that selects data based on specific criteria and performs aggregation. The execution time of this query is measured, and the function returns both the query results and the time taken in milliseconds.

```python
def query_table(measurement_name_f=measurement_name):
    with InfluxDBClient(url=url, token=token, org=org) as client:

        query_api = client.query_api()

        query =  f'''
        from(bucket: "bucket1")
          |> range(start: {start}, stop: {stop})
          |> filter(fn: (r) => r["_measurement"] == "{measurement_name}")
          |> filter(fn: (r) => r["_field"] == "{total_load_value}")
          |> filter(fn: (r) => r["AreaTypeCode"] == "CTY")
          |> aggregateWindow(every: 15m, fn: last, createEmpty: false)
          |> yield(name: "last")
        '''

        start_time = perf_counter()
        result = query_api.query_csv(query)
        diff_time = perf_counter() - start_time

        response = []
        for record in result:
            response.append({'__time': record[5], 'AreaName': record[7],
'TotalLoadValue': record[6]})

        return response[4:], int((diff_time) * 1000)
```

Code Snippet 5. The query_table function for InfluxDB.

For Apache Druid, the process remains similar, with modifications in the ingest and query functions to suit the Druid framework. Our testing utilizes a specific ingestion specification, tailored for the Druid environment.

```
INGESTION_SPEC_FOR_ALL_IN_ONE = {
    "type": "index_parallel",
    "spec": {
        "ioConfig": {
            "type": "index_parallel",
            "inputSource": {
                "type": "local",
                "baseDir": FILE_FOLDER,
                "filter": "*"
            },
            "inputFormat": {
                "type": "tsv",
                "findColumnsFromHeader": True
            },
            "appendToExisting": False
        },
        "tuningConfig": {
            "type": "index_parallel",
            "partitionsSpec": {
                "type": "dynamic"
            }
        },
        "dataSchema": {
            "dataSource": "please_specify",
            "timestampSpec": {
                "column": "\ufeffDateTime",
                "format": "auto"
            },
            "metricsSpec": [
                {
                    "type": "doubleSum",
                    "name": METRIC_NAME,
                    "fieldName": METRIC_NAME
                }
            ],
            "dimensionsSpec": {
                "dimensions": [
                    "AreaName"
                ]
            },
            "granularitySpec": {
                "queryGranularity": "none",
                "rollup": False,
                "segmentGranularity": "month"
            },
```

Code Snippet 6. Ingestion Spec Used For Druid.

The Druid version of the query table function executes SQL queries on the Druid database, using the HTTP POST method to send the query to the Druid SQL endpoint. It processes the query, receives the response in JSON format, and calculates the query duration in milliseconds.

```
def query_table(data_source_name,):

    endpoint = "/druid/v2/sql"
    http_method = "POST"

    payload = json.dumps({
        "query": "SELECT  * FROM " + data_source_name
    })

    response = requests.request(
        http_method, druid_host + endpoint, headers=headers, data=payload,
timeout=60)

    response_json = response.json()
    delta = response.elapsed.microseconds * 0.001
    return response_json, int(delta)

```

Code Snippet 7. The query_table function for Druid.

Similarly, *the ingest_data_using_file_location* function in Druid ingests data using provided file locations. It sends a POST request with a detailed ingestion specification to the Druid ingestion endpoint. The function then monitors the ingestion task's status until completion and returns the duration of the task, providing a measure of the ingestion process's efficiency.

```python
def ingest_data_using_file_location(input_file_location, ingestion_spec):
    http_method = "POST"
    endpoint = "/druid/indexer/v1/task"
    headers = {'Content-Type': 'application/json'}

    spec = copy.deepcopy(ingestion_spec)
    spec['spec']['ioConfig']['inputSource']['baseDir'] = input_file_location

    payload = json.dumps(spec)

    ingestion_task_id_response = requests.request(
        http_method, DRUID_HOST+endpoint, headers=headers, data=payload,
timeout=60)

    ingestion_task_id = json.loads(ingestion_task_id_response.text)['task']

    endpoint = f"/druid/indexer/v1/task/{ingestion_task_id}/status"

    http_method = "GET"
    payload = headers = {}

    response = requests.request(
        http_method, DRUID_HOST+endpoint, headers=headers, data=payload)

    ingestion_status = json.loads(response.text)['status']['status']

    while ingestion_status != "SUCCESS":
        response = requests.request(
            http_method, DRUID_HOST+endpoint, headers=headers, data=payload)
        ingestion_status = json.loads(response.text)['status']['status']
        time.sleep(1)

    return json.loads(response.text)['status']['duration']
```

Code Snippet 8. The function to Insert data to Apache Druid using the API.

## InfluxDB and Apache Druid Schema Structure

In InfluxDB, a dedicated time-series database, the schema is intricately designed to optimize data storage and query performance for time-series data. This schema comprises several components. The 'Measurement' is akin to a table in traditional relational databases and represents a collection of data points. For example, 'my_measurement_name' denotes the name of the measurement. 'Tags' serve as indexed metadata, crucial for efficient querying. Examples of tags include 'AreaName' and 'AreaTypeCode', both string types, which facilitate effective data querying. 'Fields' represent the actual data values, such as 'TotalLoadValue', a numeric type indicating the metric or measurement that changes over time. Lastly, the 'Timestamp' is implicitly managed by InfluxDB and can originate from the DataFrame or the server timestamp at the time of data ingestion.

| Component | Name | Data Type | Description |
|---|---|---|---|
| Measurement | my_measurement_name | - | Name of the measurement (analogous to table name in relational databases) |
| Tag | AreaName | String | Indexed metadata, used for efficiently querying data |
| Tag | AreaTypeCode | String | Indexed metadata, used for efficiently querying data |
| Field | TotalLoadValue | Numeric | Actual data value, the metric or measurement that changes over time |
| Timestamp | - | DateTime | Automatically managed by InfluxDB; either from the DataFrame or server timestamp at ingestion time |

Apache Druid, another time-series database, features a schema organization tailored for high-speed data aggregation and intricate analytical queries. It diverges slightly from InfluxDB's design. The 'DataSource' in Druid is equivalent to a table or measurement in other databases and stores sets of related data points, like 'my_measurement_name'. 'Dimensions', like InfluxDB's tags, are used for filtering, grouping, and aggregation. They include 'AreaName' and 'AreaTypeCode', both of which are string types. 'Metrics' in Druid are numerical measurements intended for analysis, with 'TotalLoadValue' (a float type) exemplifying this, supporting various metrics such as count, sum, min, and max. A unique aspect of Druid's schema is the 'Timestamp', a special column named '__time' used to partition and sort data.

| Component | Name | Data Type | Description |
|---|---|---|---|

|  |  |  |  |
|---|---|---|---|
| DataSource | my_measurement_name | - | Equivalent to a table in traditional databases; stores a set of related data points |
| Dimension | AreaName | String | Used for filtering, grouping, and aggregation. Like tags in InfluxDB |
| Dimension | AreaTypeCode | String | Used for filtering, grouping, and aggregation. Like tags in InfluxDB |
| Metric | TotalLoadValue | Float | A quantitative measurement. Druid supports various metric types like count, sum, min, max, etc. |
| Timestamp | __time | DateTime | A special column in Druid that is used to partition and sort the data |

## Performance Testing Scenarios and Measurements

To evaluate the performance of these databases, three distinct scenarios were considered. The first, 'Fresh System Installation' (shown in the diagrams as ***empty***), tests the system with an empty database following installation, establishing a baseline performance metric. The second scenario, 'Loaded Database on a New Table', involves a database laden with substantial historical data - 200 tables each holding 120 months of data, totaling approximately 80GB and 340 million entries. However, operations in this scenario are performed on a new, empty table (this is labeled as ***full-empty***). The final scenario, 'Loaded Database on an Existing Table', is like the second but involves executing operations on a table already containing 1.6 million lines of historical data table (this is labeled as ***full***).

For the TSDB the measurements were taken by either sending requests to the database and waiting for the response and timing the time difference with the python performance counter package[20] or by logging the reported time by the databases themselves. We found that the values between the 2 were almost identical.

### DRUID MEASUREMENTS

In the accompanying figure, we present a graphical representation of query performance after the incorporation of each data file. The horizontal axis (x-axis) delineates the date corresponding to each file, offering a temporal context. Vertically (y-axis), we display the duration required to execute a query on the table, which has been progressively augmented with the respective data files. To furnish a comparative perspective, we have superimposed an additional layer of data indicating the size of each file. This layered approach facilitates a comprehensive analysis of the correlation between file size and query performance over time.

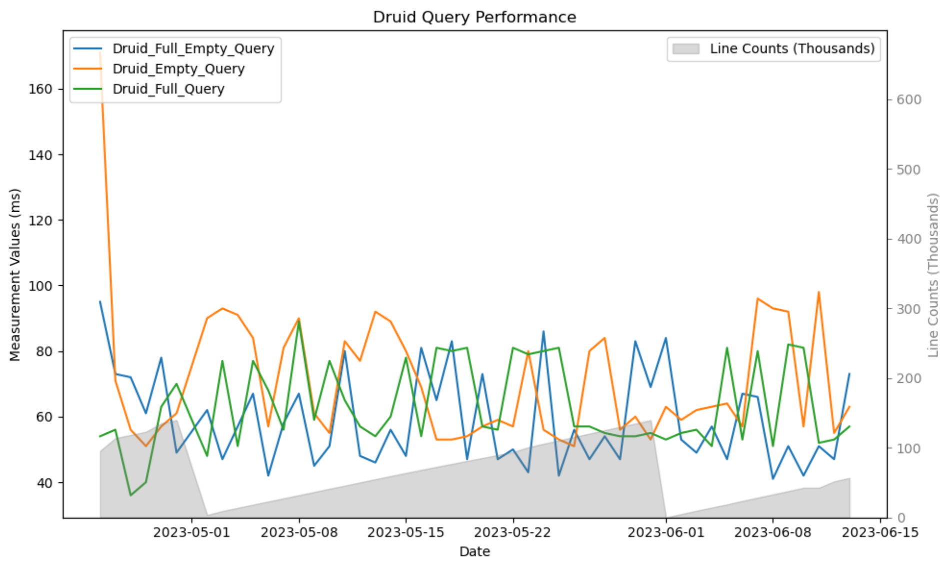


Figure 12. Druid Query Performance

The graphical representation reveals fluctuations within the line, indicative of variations in the data. However, a closer examination reveals the absence of a discernible, significant trend in these variations. Furthermore, the data does not demonstrate a consistent pattern where any specific scenario persistently outperforms or underperforms relative to the others. This suggests a lack of strong correlation or causative factors influencing the performance metrics across the different scenarios represented.

In the subsequent graphical depiction, we focus on the performance metrics pertinent to data ingestion. A discernible correlation emerges between the size of the data files and the ingestion performance. Notably, the graph exhibits distinct inflection points coinciding with the commencement of new months, where the file sizes are observed to decrease. This phenomenon introduces a step-like pattern in the performance curve. Despite these observations, it is important to note that no significant difference in ingestion performance is evident when comparing scenarios with empty databases to those with databases at or near capacity. This lack of differentiation suggests that database fullness does not markedly influence the efficiency of data ingestion under the conditions studied.

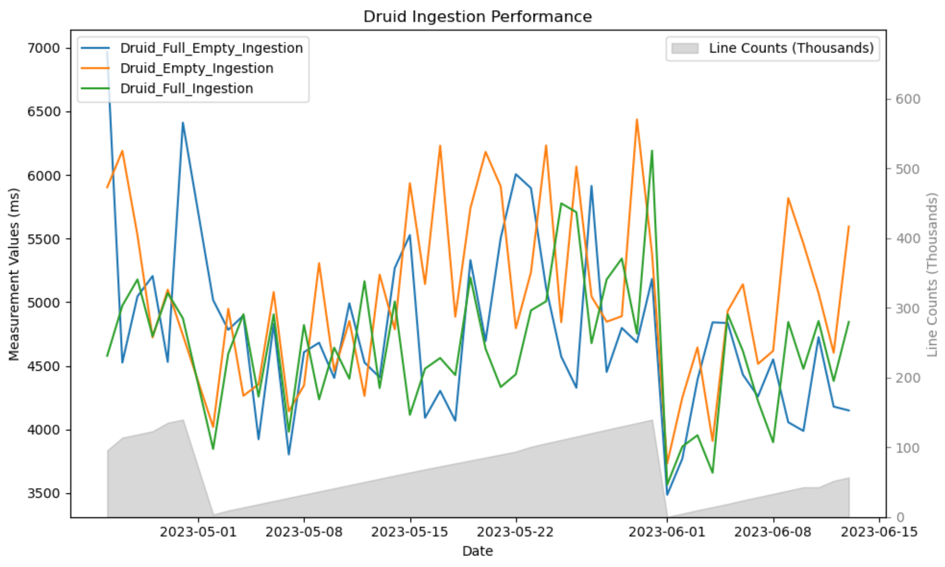


Figure 13. Druid Ingestion Performance

In the forthcoming enhancement of graphical analysis, we incorporate SQL performance metrics for a comparative evaluation with the previously examined data ingestion framework. The augmented graph elucidates a notable trend: for smaller data inputs, SQL demonstrates superior efficiency in data ingestion. However, this dynamic alters as the size of the data escalates. Beyond a certain threshold of data volume, the ingestion capabilities of the alternative system, Druid, begin to surpass those of SQL. This transition marks a critical inflection point where the relative performance advantage shifts. It becomes apparent that with increasing data sizes, the Druid system exhibits enhanced ingestion speeds, outperforming SQL. This observation underscores the scalability and efficiency of the Druid system in handling larger datasets, as compared to the SQL framework.

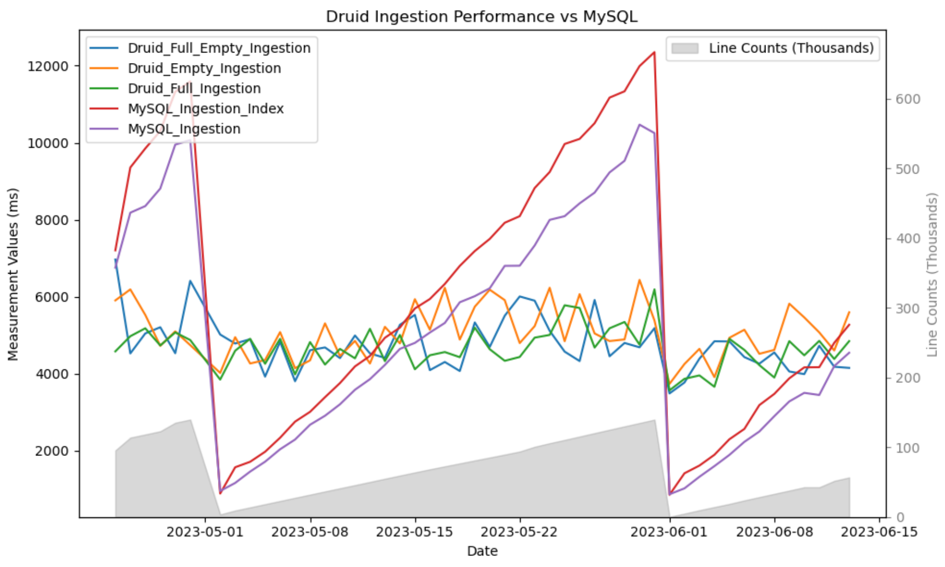


Figure 14. Druid Ingestion Performance vs MySQL

| Date | File Size_KB | Druid_Clean_Ingestion | Druid_Clean_Query | Druid_Full_Ingestion | Druid_Full_Query | Druid_Empty_Ingestion | Druid_Empty_Query | Druid_Full_Empty_Ingestion | Druid_Full_Empty_Query |
|---|---|---|---|---|---|---|---|---|---|
| 25/4 | 9113.6 | 7823 | 77 | 4579 | 54 | 5903 | 171 | 6965 | 95 |
| 26/4 | 11264 | 7908 | 74 | 4973 | 56 | 6189 | 71 | 4526 | 73 |
| 27/4 | 11264 | 8079 | 86 | 5179 | 36 | 5526 | 56 | 5046 | 72 |
| 28/4 | 12288 | 7850 | 90 | 4733 | 40 | 4723 | 51 | 5206 | 61 |
| 29/4 | 13312 | 8165 | 85 | 5072 | 63 | 5099 | 57 | 4531 | 78 |
| 30/4 | 13312 | 8340 | 93 | 4874 | 70 | 4742 | 61 | 6411 | 49 |
| 2/5 | 344 | 6176 | 97 | 3847 | 48 | 4020 | 90 | 5017 | 62 |
| 3/5 | 880 | 6369 | 98 | 4596 | 77 | 4949 | 93 | 4783 | 47 |
| 4/5 | 1331.2 | 6498 | 98 | 4906 | 51 | 4265 | 91 | 4896 | 57 |
| 5/5 | 1740.8 | 6673 | 95 | 4258 | 77 | 4355 | 84 | 3924 | 67 |
| 6/5 | 2252.8 | 6665 | 104 | 4905 | 68 | 5080 | 57 | 4829 | 42 |
| 7/5 | 2662.4 | 6827 | 102 | 3982 | 56 | 4143 | 81 | 3804 | 58 |
| 8/5 | 3072 | 6608 | 105 | 4821 | 89 | 4346 | 90 | 4608 | 67 |
| 9/5 | 3481.6 | 7045 | 99 | 4237 | 59 | 5307 | 61 | 4682 | 45 |
| 10/5 | 3891.2 | 6841 | 100 | 4643 | 77 | 4459 | 55 | 4405 | 51 |
| 11/5 | 4403.2 | 6953 | 105 | 4399 | 65 | 4852 | 83 | 4992 | 80 |
| 12/5 | 4812.8 | 7261 | 98 | 5165 | 57 | 4264 | 77 | 4527 | 48 |
| 13/5 | 5222.4 | 7518 | 101 | 4326 | 54 | 5217 | 92 | 4410 | 46 |
| 14/5 | 5632 | 7250 | 103 | 5006 | 60 | 4787 | 89 | 5269 | 56 |
| 15/5 | 6041.6 | 7480 | 100 | 4115 | 78 | 5935 | 80 | 5528 | 48 |
| 16/5 | 6451.2 | 7545 | 105 | 4478 | 54 | 5143 | 69 | 4092 | 81 |
| 17/5 | 6860.8 | 7688 | 102 | 4562 | 81 | 6230 | 53 | 4304 | 65 |
| 18/5 | 7270.4 | 7436 | 100 | 4428 | 80 | 4886 | 53 | 4069 | 83 |
| 19/5 | 7680 | 7827 | 100 | 5194 | 81 | 5740 | 54 | 5331 | 47 |
| 20/5 | 8089.6 | 7544 | 103 | 4635 | 57 | 6181 | 57 | 4695 | 73 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 21/5 | 8499.2 | 7571 | 106 | 4334 | 56 | 5910 | 59 | 5505 | 47 |
| 22/5 | 8908.8 | 7827 | 123 | 4434 | 81 | 4795 | 57 | 6006 | 50 |
| 23/5 | 9523.2 | 8164 | 109 | 4936 | 79 | 5236 | 80 | 5896 | 43 |
| 24/5 | 10035.2 | 7736 | 101 | 5007 | 80 | 6232 | 56 | 5113 | 86 |
| 25/5 | 11264 | 8114 | 105 | 5777 | 81 | 4843 | 53 | 4573 | 42 |
| 26/5 | 11264 | 8138 | 109 | 5707 | 57 | 6066 | 51 | 4328 | 56 |
| 27/5 | 12288 | 8150 | 99 | 4678 | 57 | 5045 | 80 | 5914 | 47 |
| 28/5 | 12288 | 8437 | 109 | 5179 | 55 | 4847 | 84 | 4452 | 54 |
| 29/5 | 12288 | 8132 | 99 | 5343 | 54 | 4890 | 56 | 4798 | 47 |
| 30/5 | 13312 | 7975 | 110 | 4754 | 54 | 6436 | 60 | 4685 | 83 |
| 31/5 | 13312 | 8111 | 118 | 6191 | 55 | 5380 | 53 | 5183 | 69 |
| 1/6 | 28 | 5946 | 97 | 3567 | 53 | 3734 | 63 | 3487 | 84 |
| 2/6 | 460 | 6229 | 96 | 3866 | 55 | 4251 | 59 | 3771 | 53 |
| 3/6 | 920 | 6250 | 93 | 3955 | 56 | 4645 | 62 | 4390 | 49 |
| 4/6 | 1433.6 | 6691 | 96 | 3660 | 51 | 3910 | 63 | 4842 | 57 |
| 5/6 | 1843.2 | 6566 | 94 | 4905 | 81 | 4932 | 64 | 4837 | 47 |
| 6/6 | 2252.8 | 6750 | 109 | 4617 | 53 | 5141 | 57 | 4431 | 67 |
| 7/6 | 2764.8 | 6688 | 97 | 4223 | 80 | 4516 | 96 | 4260 | 66 |
| 8/6 | 3174.4 | 6831 | 96 | 3900 | 51 | 4618 | 93 | 4551 | 41 |
| 9/6 | 3584 | 6825 | 97 | 4845 | 82 | 5818 | 92 | 4057 | 51 |
| 10/6 | 4096 | 6849 | 95 | 4477 | 81 | 5461 | 57 | 3989 | 42 |
| 11/6 | 4096 | 6993 | 99 | 4852 | 52 | 5072 | 98 | 4726 | 51 |
| 12/6 | 4915.2 | 7464 | 95 | 4381 | 53 | 4603 | 55 | 4180 | 47 |
| 13/6 | 5427.2 | 7395 | 104 | 4846 | 57 | 5595 | 63 | 4150 | 73 |
| **Average** | **6339.820408** | **7310.22449** | **99.51020408** | **4660.142857** | **63.30612245** | **5067.693878** | **71.16326531** | **4754.571429** | **59.24489796** |

## INFLUX MEASUREMENTS

In the provided analysis, we delve into the query performance characteristics of InfluxDB, particularly focusing on the impact of database and table occupancy levels. The data distinctly indicates a variation in performance contingent upon whether the database and the table are empty or full. This distinction is evident in two separate scenarios: one where the database is full, and another where the table within the database is full. Quantitatively, performance degradation can be characterized by an average increase in response time. Specifically, there is an average increment of 3 milliseconds attributed to the presence of a fully populated database, and an additional increase of 3 milliseconds when the table itself is full. This cumulative impact of 6 milliseconds underscores the sensitivity of InfluxDB's query performance to both database and table occupancy levels, providing valuable insights into its operational efficiency under varying data storage conditions.

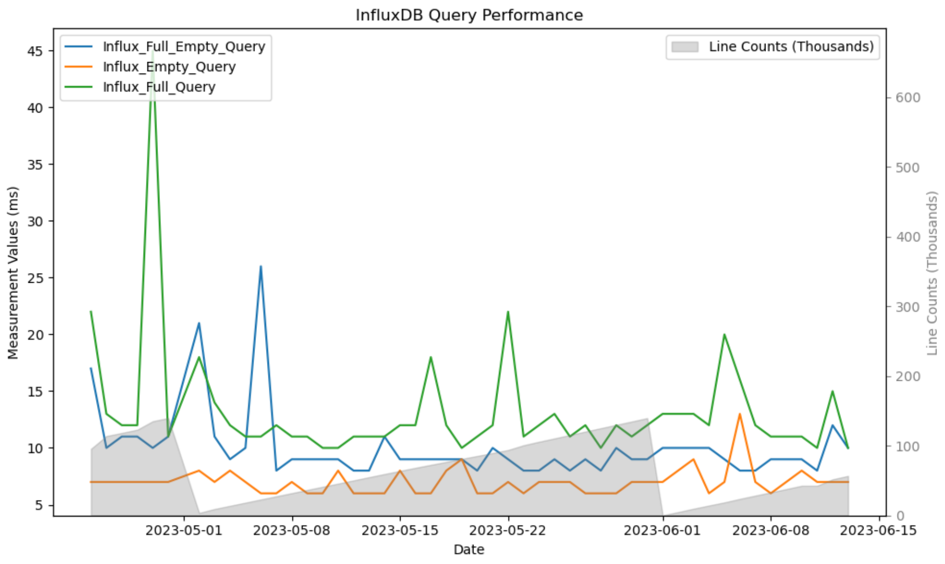


Figure 15. InfluxDB Query Performance

In our analysis of the ingestion performance, a strikingly linear relationship emerges, aligning precisely with the size of the files being ingested. This correlation is represented by a consistently straight line in the graphical depiction, indicating a direct proportionality between file size and ingestion performance metrics. However, it is noteworthy that this relationship appears invariant across different scenarios. Regardless of the varying conditions or parameters under which ingestion occurs, there is no perceivable deviation in the performance trend. This uniformity suggests a robustness in the ingestion process, where performance is predominantly dictated by file size, unaffected by other potential variables or operational scenarios.

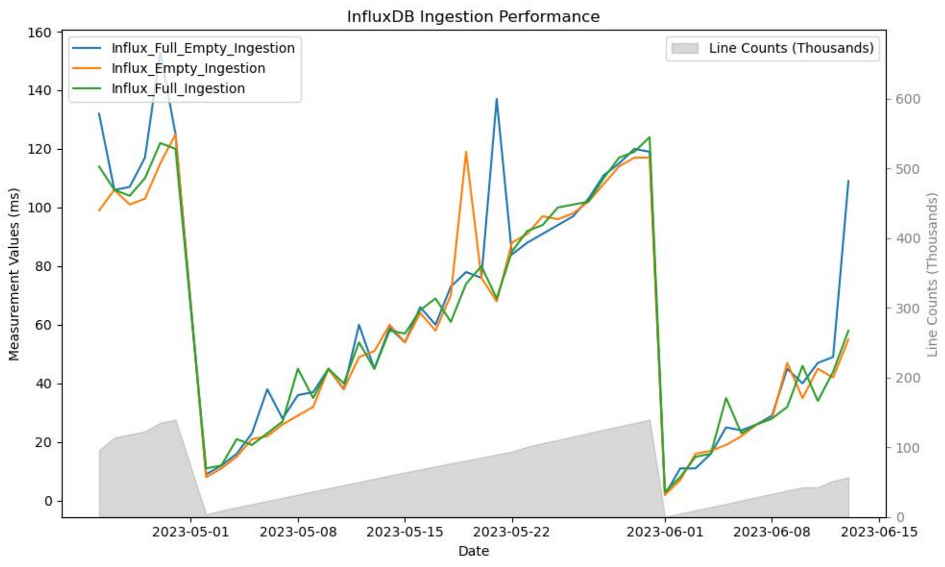


Figure 16. InfluxDB Ingestion Performance

With the introduction of SQL into our comparative analysis, a significant shift in the performance landscape is observed. The graphical representation now reveals a marked disparity in ingestion performance between SQL and InfluxDB. This difference is not marginal but rather spans two orders of magnitude. Specifically, SQL's ingestion times hover around the 12-second mark. In stark contrast, InfluxDB demonstrates a more efficient performance, with ingestion times averaging around 120 milliseconds. This substantial disparity highlights the vastly superior ingestion efficiency of InfluxDB over SQL in the examined scenarios, providing a clear indication of its potential for high-speed data handling and processing capabilities.

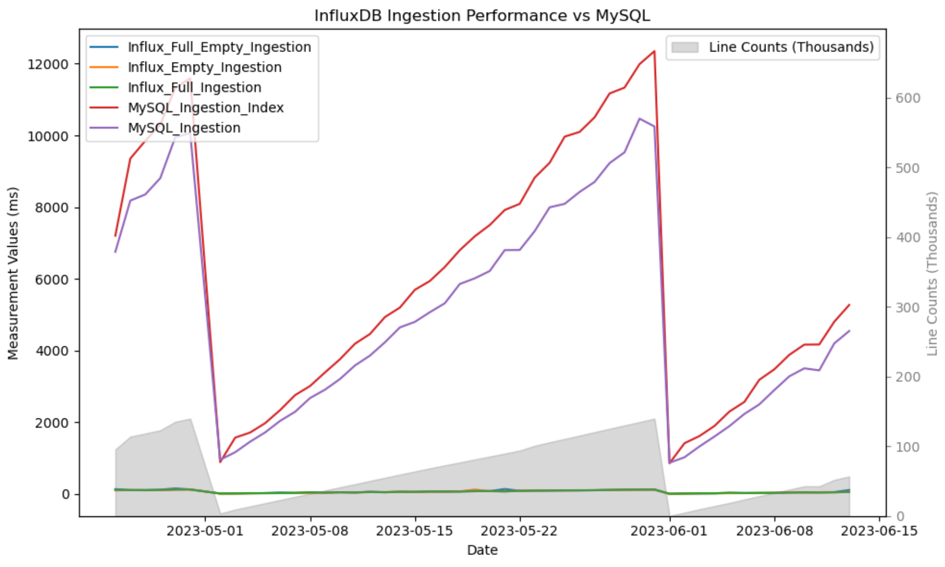


Figure 17. InfluxDB Ingestion Performance vs MySQL

| Date | FileSize_KB | Influx_Full_Ingestion | Influx_Full_Query | Influx_Empty_Ingestion | Influx_Empty_Query | Influx_Full_Empty_Ingestion | Influx_Full_Empty_Query |
|---|---|---|---|---|---|---|---|
| 25/4 | 9113.6 | 114 | 22 | 99 | 7 | 132 | 17 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 26/4 | 11264 | 106 | 13 | 106 | 7 | 106 | 10 |
| 27/4 | 11264 | 104 | 12 | 101 | 7 | 107 | 11 |
| 28/4 | 12288 | 110 | 12 | 103 | 7 | 117 | 11 |
| 29/4 | 13312 | 122 | 45 | 115 | 7 | 153 | 10 |
| 30/4 | 13312 | 120 | 11 | 125 | 7 | 125 | 11 |
| 2/5 | 344 | 11 | 18 | 8 | 8 | 9 | 21 |
| 3/5 | 880 | 12 | 14 | 11 | 7 | 12 | 11 |
| 4/5 | 1331.2 | 21 | 12 | 15 | 8 | 16 | 9 |
| 5/5 | 1740.8 | 19 | 11 | 21 | 7 | 23 | 10 |
| 6/5 | 2252.8 | 23 | 11 | 22 | 6 | 38 | 26 |
| 7/5 | 2662.4 | 27 | 12 | 26 | 6 | 28 | 8 |
| 8/5 | 3072 | 45 | 11 | 29 | 7 | 36 | 9 |
| 9/5 | 3481.6 | 35 | 11 | 32 | 6 | 37 | 9 |
| 10/5 | 3891.2 | 45 | 10 | 45 | 6 | 45 | 9 |
| 11/5 | 4403.2 | 40 | 10 | 38 | 8 | 38 | 9 |
| 12/5 | 4812.8 | 54 | 11 | 49 | 6 | 60 | 8 |
| 13/5 | 5222.4 | 45 | 11 | 51 | 6 | 45 | 8 |
| 14/5 | 5632 | 58 | 11 | 60 | 6 | 59 | 11 |
| 15/5 | 6041.6 | 57 | 12 | 54 | 8 | 54 | 9 |
| 16/5 | 6451.2 | 65 | 12 | 64 | 6 | 66 | 9 |
| 17/5 | 6860.8 | 69 | 18 | 58 | 6 | 60 | 9 |
| 18/5 | 7270.4 | 61 | 12 | 70 | 8 | 73 | 9 |
| 19/5 | 7680 | 74 | 10 | 219 | 9 | 78 | 9 |
| 20/5 | 8089.6 | 80 | 11 | 76 | 6 | 76 | 8 |
| 21/5 | 8499.2 | 69 | 12 | 68 | 6 | 137 | 10 |
| 22/5 | 8908.8 | 85 | 22 | 88 | 7 | 84 | 9 |
| 23/5 | 9523.2 | 92 | 11 | 91 | 6 | 88 | 8 |
| 24/5 | 10035.2 | 94 | 12 | 97 | 7 | 91 | 8 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 25/5 | 11264 | 100 | 13 | 96 | 7 | 94 | 9 |
| 26/5 | 11264 | 101 | 11 | 98 | 7 | 97 | 8 |
| 27/5 | 12288 | 102 | 12 | 102 | 6 | 103 | 9 |
| 28/5 | 12288 | 110 | 10 | 108 | 6 | 111 | 8 |
| 29/5 | 12288 | 117 | 12 | 114 | 6 | 115 | 10 |
| 30/5 | 13312 | 119 | 11 | 117 | 7 | 120 | 9 |
| 31/5 | 13312 | 124 | 12 | 117 | 7 | 119 | 9 |
| 1/6 | 28 | 3 | 13 | 2 | 7 | 2 | 10 |
| 2/6 | 460 | 8 | 13 | 7 | 8 | 11 | 10 |
| 3/6 | 920 | 15 | 13 | 16 | 9 | 11 | 10 |
| 4/6 | 1433.6 | 16 | 12 | 17 | 6 | 16 | 10 |
| 5/6 | 1843.2 | 35 | 20 | 19 | 7 | 25 | 9 |
| 6/6 | 2252.8 | 23 | 16 | 22 | 13 | 24 | 8 |
| 7/6 | 2764.8 | 26 | 12 | 26 | 7 | 26 | 8 |
| 8/6 | 3174.4 | 28 | 11 | 28 | 6 | 29 | 9 |
| 9/6 | 3584 | 32 | 11 | 47 | 7 | 45 | 9 |
| 10/6 | 4096 | 46 | 11 | 35 | 8 | 40 | 9 |
| 11/6 | 4096 | 34 | 10 | 45 | 7 | 47 | 8 |
| 12/6 | 4915.2 | 44 | 15 | 42 | 7 | 49 | 12 |
| 13/6 | 5427.2 | 58 | 10 | 55 | 7 | 109 | 10 |
| **Average** | **6339.820408** | **61.18367347** | **13.2244898** | **62.32653061** | **7** | **65.02040816** | **10** |

## Comparing the Results

To effectively represent the comparison of two time series databases (TSDBs) in graphical form, focusing on InfluxDB and another TSDB, we can create two distinct types of plots:

Query Performance Graph: This graph will compare the query performance of InfluxDB with Apache Druid. Here, we can use a line graph or bar chart where the Y-axis represents the time taken for queries and the X-axis represents different query scenarios. InfluxDB's line or bars would be consistently lower on the graph, indicating better (faster) performance.

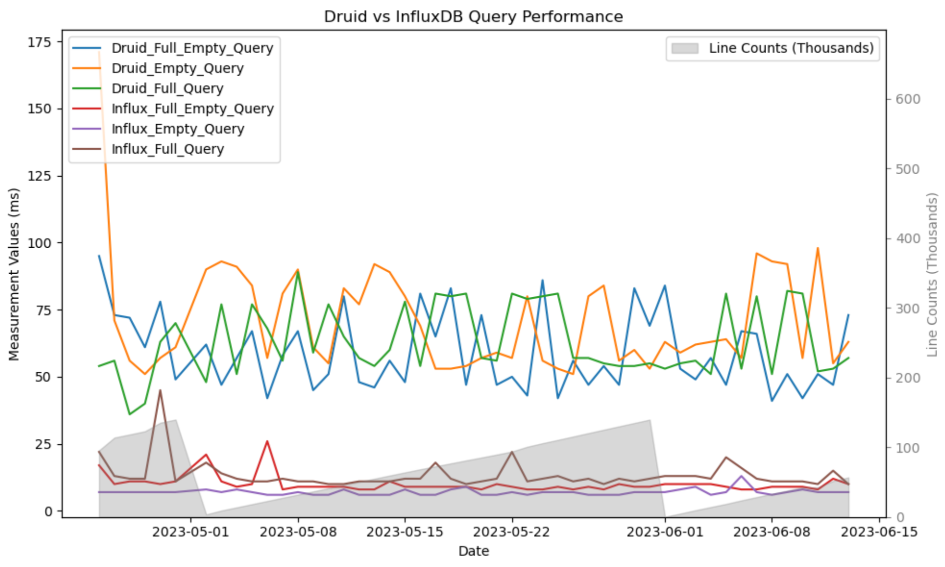


Figure 18. Druid vs InfluxDB Query Performance

Ingestion Performance Graph: Like the query performance graph, this one will focus on the ingestion performance, which is the speed and efficiency with which the databases can ingest or input data. Again, a line graph or bar chart would work well, with the Y-axis showing the time taken for data ingestion and the X-axis representing various data ingestion scenarios. InfluxDB would again demonstrate superiority by having lower values on the graph.

MySQL joins the comparison alongside Apache Druid and InfluxDB. Throughout the range of tests conducted, InfluxDB consistently outperforms the others, maintaining the lead in speed. Apache Druid, on average, ranks second in terms of performance, while MySQL typically falls behind, occupying the last position in this comparative study.

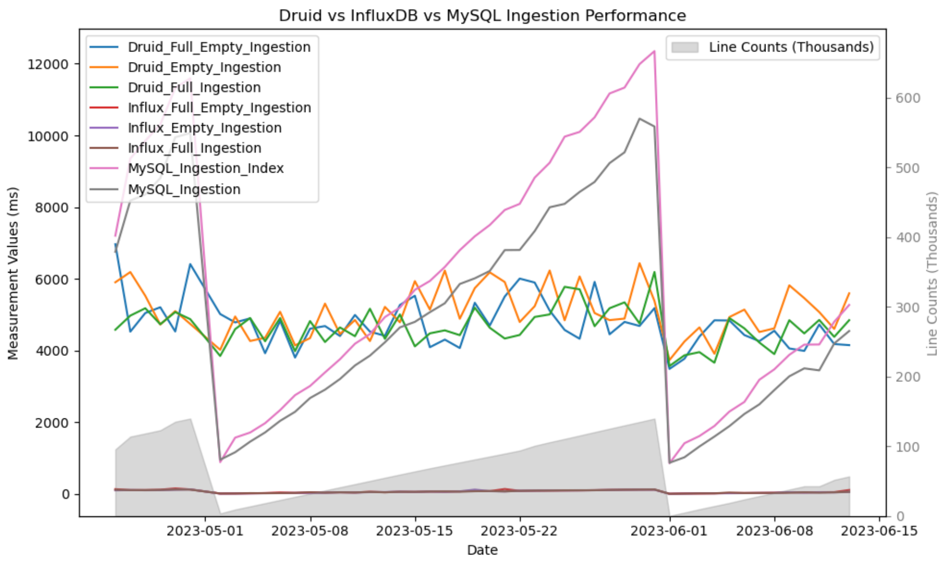


Figure 19. Druid vs InfluxDB Ingestion Performance

Below we have also the data from the table above.

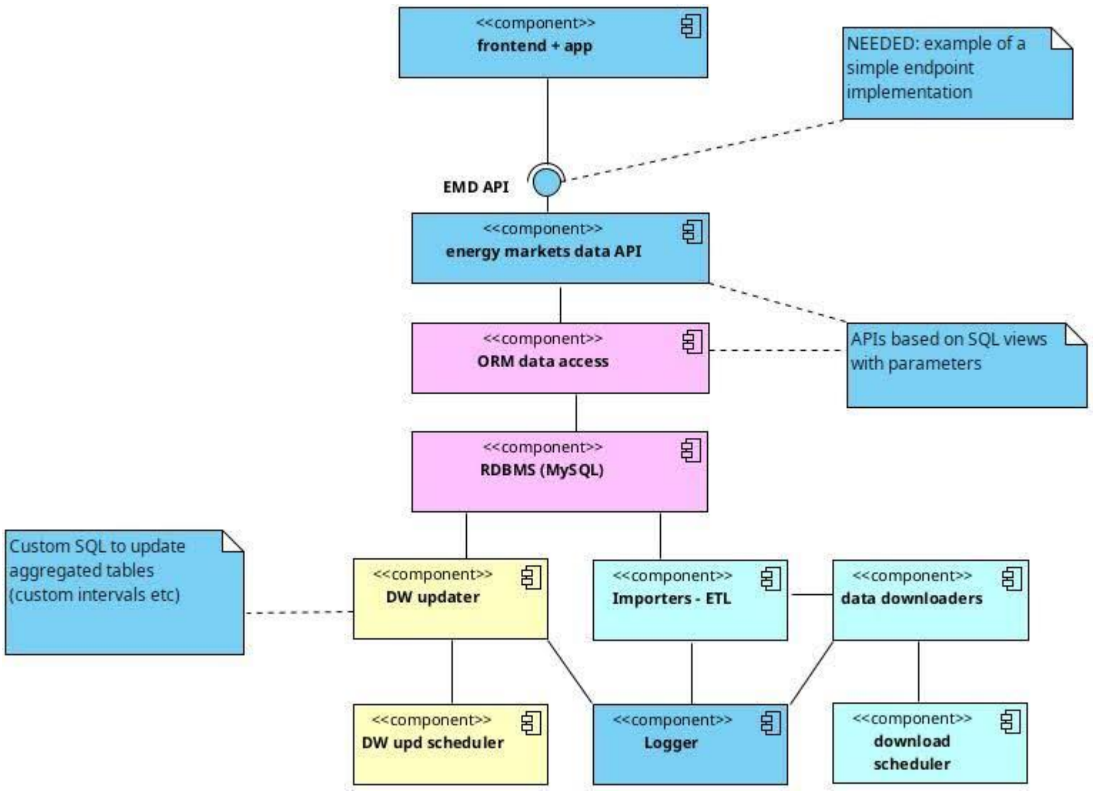
| Date | FileSize_KB | MySQL_Measurement | Druid_Full_Ingestion | Druid_Full_Query | Influx_Full_Ingestion | Influx_Full_Query |
|---|---|---|---|---|---|---|
| 25/4 | 9113.6 | 6753.5 | 4579 | 54 | 114 | 22 |
| 26/4 | 11264 | 8181.6 | 4973 | 56 | 106 | 13 |
| 27/4 | 11264 | 8353.1 | 5179 | 36 | 104 | 12 |
| 28/4 | 12288 | 8808.4 | 4733 | 40 | 110 | 12 |
| 29/4 | 13312 | 9949.3 | 5072 | 63 | 122 | 45 |
| 30/4 | 13312 | 10066.3 | 4874 | 70 | 120 | 11 |
| 2/5 | 344 | 953.4 | 3847 | 48 | 11 | 18 |
| 3/5 | 880 | 1161.89 | 4596 | 77 | 12 | 14 |
| 4/5 | 1331.2 | 1457.2 | 4906 | 51 | 21 | 12 |
| 5/5 | 1740.8 | 1714.5 | 4258 | 77 | 19 | 11 |
| 6/5 | 2252.8 | 2036.6 | 4905 | 68 | 23 | 11 |
| 7/5 | 2662.4 | 2289.7 | 3982 | 56 | 27 | 12 |
| 8/5 | 3072 | 2673.6 | 4821 | 89 | 45 | 11 |
| 9/5 | 3481.6 | 2907.6 | 4237 | 59 | 35 | 11 |
| 10/5 | 3891.2 | 3204.8 | 4643 | 77 | 45 | 10 |
| 11/5 | 4403.2 | 3582.7 | 4399 | 65 | 40 | 10 |
| 12/5 | 4812.8 | 3857.2 | 5165 | 57 | 54 | 11 |
| 13/5 | 5222.4 | 4228.7 | 4326 | 54 | 45 | 11 |
| 14/5 | 5632 | 4644 | 5006 | 60 | 58 | 11 |
| 15/5 | 6041.6 | 4799.1 | 4115 | 78 | 57 | 12 |
| 16/5 | 6451.2 | 5068.6 | 4478 | 54 | 65 | 12 |
| 17/5 | 6860.8 | 5318 | 4562 | 81 | 69 | 18 |
| 18/5 | 7270.4 | 5855.4 | 4428 | 80 | 61 | 12 |
| 19/5 | 7680 | 6013.3 | 5194 | 81 | 74 | 10 |
| 20/5 | 8089.6 | 6218.5 | 4635 | 57 | 80 | 11 |
| 21/5 | 8499.2 | 6801.1 | 4334 | 56 | 69 | 12 |
| 22/5 | 8908.8 | 6804.2 | 4434 | 81 | 85 | 22 |
| 23/5 | 9523.2 | 7330.5 | 4936 | 79 | 92 | 11 |
| 24/5 | 10035.2 | 7995 | 5007 | 80 | 94 | 12 |
| 25/5 | 11264 | 8090.8 | 5777 | 81 | 100 | 13 |
| 26/5 | 11264 | 8423.5 | 5707 | 57 | 101 | 11 |
| 27/5 | 12288 | 8702.3 | 4678 | 57 | 102 | 12 |
| 28/5 | 12288 | 9227.5 | 5179 | 55 | 110 | 10 |
| 29/5 | 12288 | 9529.1 | 5343 | 54 | 117 | 12 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 30/5 | 13312 | 10468.7 | 4754 | 54 | 119 | 11 |
| 31/5 | 13312 | 10248.2 | 6191 | 55 | 124 | 12 |
| 1/6 | 28 | 868.1 | 3567 | 53 | 3 | 13 |
| 2/6 | 460 | 1021.3 | 3866 | 55 | 8 | 13 |
| 3/6 | 920 | 1322.6 | 3955 | 56 | 15 | 13 |
| 4/6 | 1433.6 | 1599.39 | 3660 | 51 | 16 | 12 |
| 5/6 | 1843.2 | 1890.1 | 4905 | 81 | 35 | 20 |
| 6/6 | 2252.8 | 2229.4 | 4617 | 53 | 23 | 16 |
| 7/6 | 2764.8 | 2497.7 | 4223 | 80 | 26 | 12 |
| 8/6 | 3174.4 | 2896.7 | 3900 | 51 | 28 | 11 |
| 9/6 | 3584 | 3279.4 | 4845 | 82 | 32 | 11 |
| 10/6 | 4096 | 3502.2 | 4477 | 81 | 46 | 11 |
| 11/6 | 4096 | 3446 | 4852 | 52 | 34 | 10 |
| 12/6 | 4915.2 | 4196.7 | 4381 | 53 | 44 | 15 |
| 13/6 | 5427.2 | 4543.9 | 4846 | 57 | 58 | 10 |
| **Average** | **6339.820408** | **5041.048571** | **4660.142857** | **63.30612245** | **61.18367347** | **13.2244898** |

# 6. Integrating Time Series Databases in Distributed Architectures: Enhancing legacy Systems

Having explained the performance improvements gained by Time Series Databases (TSDBs) over traditional MySQL databases, this thesis now pivots to explore their integration within distributed architectures. Specifically, the discourse will focus on pragmatic methodologies for upgrading existing operational systems employing TSDBs. This investigation delves into two distinct upgrade paths: firstly, the substitution of our current data warehouse with a TSDB, and secondly, a more holistic approach where both the data warehouse and the existing relational database are replaced with a unified TSDB solution.

The following section outlines the comprehensive legacy software architecture deployed for the acquisition, processing, and distribution of energy market data. The architecture is designed to facilitate robust data management practices and to provide a streamlined interface for front-end applications to interact with the underlying data structures. The system is organized into various components, each tailored to handle specific aspects of the data lifecycle.



<<component>> frontend + app
NEEDED: example of a simple endpoint implementation
EMD API
<<component>> energy markets data API
<<component>> ORM data access
APIs based on SQL views with parameters
<<component>> RDBMS (MySQL)
Custom SQL to update aggregated tables (custom intervals etc)
<<component>> DW updater
<<component>> Importers - ETL
<<component>> data downloaders
<<component>> DW upd scheduler
<<component>> Logger
<<component>> download scheduler

Figure 20 The Legacy System

### Front-End Application Interface

At the top of the architecture resides the Front-End Application Interface, which provides an interactive user interface for clients. It is engineered to communicate with the Energy Markets Data API, which acts as a mediator between the user-facing side of the system and the data management layers.

### Energy Markets Data API (EMD API)

The EMD API serves as a gateway, allowing the front-end application to request and receive energy market data. This API is pivotal in abstracting the complexity of the underlying data management processes, providing a simplified and coherent data access layer for the application.

## Data Management Components

### Object-Relational Mapping (ORM) Data Access:

This component utilizes an ORM framework to facilitate interaction with the database. It translates data between the incompatible type systems of relational databases and object-oriented programming languages, thereby streamlining database interactions.

### Relational Database Management System (RDBMS - MySQL):

At the heart of the data storage mechanism is the RDBMS, with MySQL employed as the database system. It is responsible for the secure and efficient storage of structured data.

### APIs Based on SQL Views with Parameters:

The APIs are augmented with parameterized SQL views, enabling dynamic and flexible data querying capabilities. This feature allows users to tailor their data requests to specific needs and contexts.

## Data Processing and Warehousing

Special emphasis should be put on the complexity and challenges associated with the Data Warehousing (DW) component. This aspect significantly influences our preference for integrating a TSDB into our system. The DW component, as it currently stands, presents several drawbacks: it is computationally intensive, prone to errors when managed manually, and imposes a performance penalty that hasn't been adequately measured in the rest of this paper. These issues further weaken the case for a purely relational solution. One of the critical limitations of traditional DW is its rigidity in data aggregation. For instance, aggregating data to a specific time window (such as 15 or 30 minutes) is notably slower and lacks flexibility. In contrast, a TSDB allows for efficient and dynamic aggregation over various time windows. With a conventional DW, decisions about data aggregation need to be made in advance. Any subsequent changes require time-consuming migration operations. Moreover, time series databases inherently accommodate updates to individual values or specific time ranges, which is an essential feature for our system's efficiency and adaptability. This capability further underscores the advantages of a TSDB over traditional data warehousing solutions.

### Data Warehouse Updater (DW Updater):

A critical component in the data warehousing segment is the DW Updater. It is tasked with the processing and cleansing of raw, 'dirty' data, transforming it into a refined format suitable for analysis and storage.

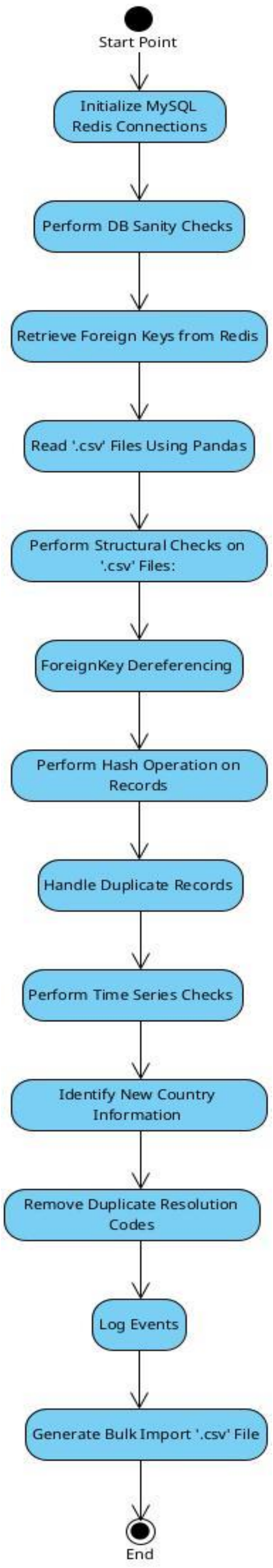
### Data Warehouse Update Scheduler (DW Update Scheduler):

The DW Update Scheduler automates the timing of data updates, ensuring that the data warehouse maintains the most current and accurate data without manual intervention.

### Custom SQL for Aggregated Tables:

Custom SQL scripts are utilized to periodically update aggregated tables within the data warehouse. These scripts are tailored to handle custom intervals and specific aggregation requirements, central to maintaining summary data for expedient access.

## Data Importation and Logging



Start Point
Initialize MySQL Redis Connections
Perform DB Sanity Checks
Retrieve Foreign Keys from Redis
Read '.csv' Files Using Pandas
Perform Structural Checks on '.csv' Files:
ForeignKey Dereferencing
Perform Hash Operation on Records
Handle Duplicate Records
Perform Time Series Checks
Identify New Country Information
Remove Duplicate Resolution Codes
Log Events
Generate Bulk Import '.csv' File
End

Figure 21. Importer Module Activity Diagram

The importer module is responsible for storing the data located in '.csv' files into the local MySQL database and simultaneously performing data cleansing, pre-processing and post-processing operations. The python class performing the above operations requires connection to a MySQL server, a Redis server and several python libraries such as pandas, numpy e.t.c.

The operations performed by the module are the following:

- Performs DB sanity checks on the foreign keys of the database to verify uniqueness of each foreign key utilized.
- Initializes connections to the Redis and MySQL servers.
- Retrieves all the foreign keys located as string key-value pairs in the Redis server and performs check on them to verify equality with the corresponding database foreign keys.
- Utilizes pandas library to read the '.csv' files and store them in a pandas dataframe.
- Performs structural checks on the '.csv' files, such as expected vs current headers equality and removes all "corrupted" records not having proper number fields from the '.csv'.
- Utilizes the foreign key dictionaries to dereference the foreign keys from strings to integers, as they reside within the mapping database tables. In case a foreign key is missing, update both the MySQL table and the Redis tables
- Perform a hash operation for the given columns of a record, in order to easily distinguish the records between them and identical records already residing within the import database table.
- Keep only the most recent records in case duplicates hash exist and also erase from the database records already existing with the same hash values. The query utilized to erase the records already existing within the db is very large and time consuming.
- "Delete from <table> where RowHash IN (….) ", contains up to milions of records
- Performs time series checks. Splits the input data per unique time series for each dataset's specified dimensions (country, production type e.t.c.) and identifies duplicates and time gaps, based on each series resolution.
- Performs checks to identify new country information in the '.csv' file.
- Removes duplicate records of multiple resolution codes.
- Logs above events
- Generates a .csv file to utilize with an SQL query and bulk import the data "LOAD DATA LOCAL INFILE 'generated_csv_name';"

This importer module is time consuming and utilizes a large part of the memory, given that time series operations and deletion/insertion tasks occur on a large amount of data.

### Importers - ETL (Extract, Transform, Load):

The importer module serves a critical function in integrating data from `.csv` files into a local MySQL database while simultaneously managing various data operations such as cleansing, pre-processing, and post-processing. This module operates through a Python class, which requires connections to a MySQL server and a Redis server, and utilizes several Python libraries, including pandas and NumPy. Its primary role involves performing database sanity checks to ensure the uniqueness of each foreign key. It also initializes connections to Redis and MySQL servers and manages foreign keys by retrieving them from the Redis server (where they are stored as string key-value pairs) and ensuring they are consistent with the database foreign keys.

The module uses pandas to read `.csv` files and store the data in dataframes. It conducts structural checks on these files, comparing expected headers against current ones and discarding records with incorrect number fields. It also converts foreign keys from strings to integers to match their format in the database, updating MySQL and Redis tables when a foreign key is missing. Another significant function is the hashing of specific record columns, which helps differentiate new records from existing ones in the import database table. This process involves retaining only the most recent records when duplicate hashes are found and removing existing database records with the same hash values using a large and time-consuming query:

```
Delete from <table> where RowHash IN (...)
```

Additionally, the module performs time series analysis by splitting data per unique time series based on dataset dimensions like country and production type. It identifies duplicates and time gaps according to the series resolution and checks for new country information in `.csv` files. It also removes duplicate records that have multiple resolution codes and logs all these events. Finally, the module prepares for data import by generating a `.csv` file used for bulk data import with the SQL command:

```
LOAD DATA LOCAL INFILE ‘generated_csv_name’;
```

Despite its crucial role, the importer module is time-intensive and consumes a significant portion of memory, given the large scale of its time series operations and the extensive data deletion and insertion tasks it performs.

### Data Downloaders and Download Scheduler:

These components manage the downloading of data from external APIs or services and schedule these operations to maintain a steady and up-to-date flow of data into the system.

## Logger:

A Logger component is incorporated to chronicle system events and errors. This utility is vital for the ongoing monitoring, troubleshooting, and optimization of the system's operations.

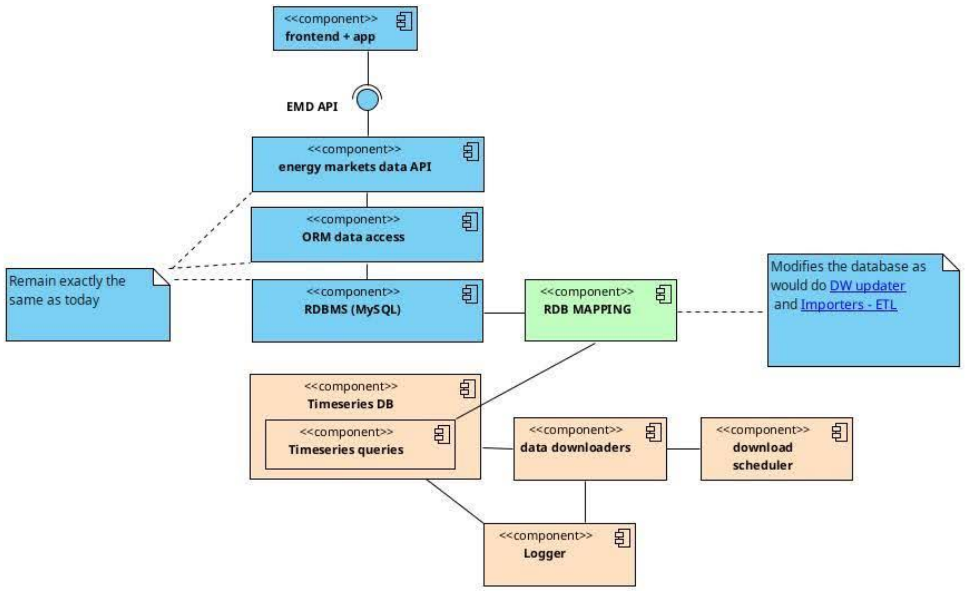
## System Interaction and Integration

The architecture is designed to promote seamless interaction between its constituent components. The front-end application interfaces with the EMD API, which in turn interacts with both the ORM data access and the parameterized SQL views to retrieve and manipulate data stored in the MySQL RDBMS. The data warehousing components ensure that data is processed and stored in a manner that facilitates efficient retrieval and analysis. The importers and downloaders work in tandem with the data processing components to maintain a continuous and automated data flow. The entire process is underpinned by the Logger, which maintains a critical oversight of system performance and integrity.

This architecture represents a robust, scalable solution for managing energy market data, designed to support high availability and responsive data access for front-end applications. The modular design ensures that each component can be independently maintained or upgraded, thereby future proofing the system.

## System architecture progression

The first significant change we will implement in this updated architecture is the addition of a timeseries database as the data warehousing component. This addition plays a pivotal role in addressing the growing complexity of energy market data and the need for advanced analytical capabilities. By incorporating a timeseries database, we can forego the heavy in engineering time process of cleaning the data and instead we can pass them through the TSDB system and pass them along.



<<component>> frontend + app
EMD API
<<component>> energy markets data API
<<component>> ORM data access
<<component>> RDBMS (MySQL)
<<component>> RDB MAPPING
Remain exactly the same as today
Modifies the database as would do DW updater and Importers - ETL
<<component>> Timeseries DB
<<component>> Timeseries queries
<<component>> data downloaders
<<component>> download scheduler
<<component>> Logger

Figure 22 Replacement of the Data Warehousing Component with TSDB

### RDB Mapping:

The updated architecture introduces RDB Mapping, an intermediary layer that modifies the database in a similar fashion to the DW Updater and the ETL processes of the original system. This component suggests a refined approach to data transformation and integration, providing a more sophisticated mechanism to ensure data consistency and integrity between the relational database and the new Timeseries DB.

### Timeseries DB:

The timeseries database component will play a crucial role in processing and cleaning the data with the same goal and efficacy as the traditional data warehousing layer. It acts as a robust intermediary layer that not only stores and organizes time-series data efficiently but also performs vital data preparation tasks to ensure that the data is of high quality and ready for advanced analytics. One of its key advantages is its reliability and accuracy in data processing and cleaning. By leveraging a specialized timeseries database, the system mitigates the risk of introducing bugs or errors that might occur if these tasks were handled manually or through custom-built solutions. This ensures that the data preparation process is not only efficient but also dependable, safeguarding the integrity of the data for critical decision-making processes in the energy market.

### Maintained Components

Notably, the update mandates that certain components, particularly the ORM Data Access and RDBMS (MySQL), "Remain exactly the same as today." This instruction indicates a deliberate decision to preserve stability and continuity in certain areas of the system while evolving others. It underscores the system's foundational reliability and the strategic focus of the enhancements on expanding capabilities rather than overhauling well-functioning elements.

### Conclusion

The updated architecture reflects a strategic evolution focused on specialized data handling and advanced analytical capabilities. By introducing a Timeseries DB and dedicated querying mechanisms, the system is now better equipped to handle the intricacies of energy market data. The inclusion of RDB Mapping indicates a commitment to sophisticated data processing techniques, ensuring that the system can maintain its core functionality while expanding to meet the demands of complex data scenarios.

## Architecture without a relational DB

In the latest iteration of the system architecture, there are further refinements and modifications which focus on the introduction of a timeseries database and the adjustments in the API layer to accommodate this change. Below is a description of the differences compared to the previous architecture.

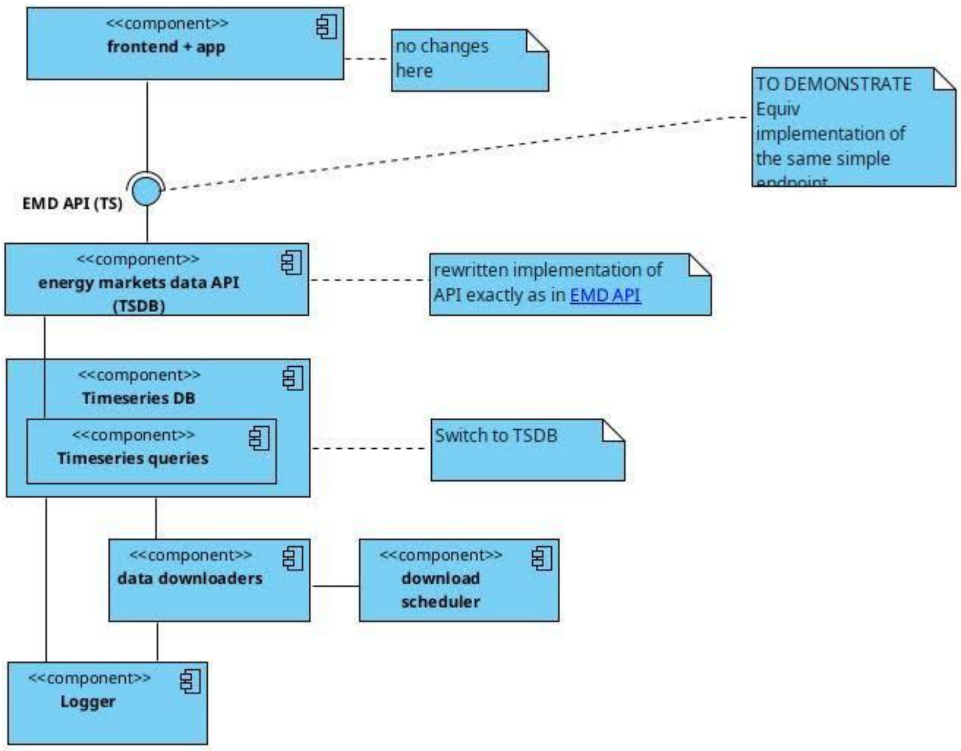


Figure 23 The Architecture Void of a relational DB

## Timeseries Database Integration

The architecture has undergone significant refinement with the integration of a Time Series Database (TSDB). This shift represents a substantial departure from our previous architecture, which relied on a traditional relational database system. With this change, in addition to the benefits we had previously achieved, we are now experiencing faster query response times and reduced duplicate data storage. However, it's worth noting that adapting to the TSDB requires us to rewrite a significant portion of the API layer to interact with the TSDB, as opposed to the relational database.

## API Layer Evolution

The Energy Markets Data API has been restructured, signifying a substantial evolution from its predecessor. The new EMD API (TS) is a rewritten implementation, indicating that while the API's functionality remains consistent with the previous version—serving the same endpoints—the underlying operations have been adapted to leverage the capabilities of the timeseries database. This alteration ensures that the API's performance is optimized for the new data structure without compromising the endpoints' expected behavior.

## Consistency in Frontend Interaction

In the realm of frontend interaction, the system architecture maintains a steady approach with no changes to the Frontend + App component. This decision implies a design philosophy where enhancements to the system's backend—such as the integration of the timeseries database—do not adversely impact the existing frontend application. Such an approach reduces the need for frontend redevelopment and ensures a consistent user experience. Despite significant backend changes, the API maintains its ability to serve the frontend application as before. This demonstrates a commitment to backward compatibility and service continuity, which is essential for system reliability and stakeholder confidence.

The further refined architecture underscores a strategic decision to enhance the system's data handling capabilities with a focus on timeseries data while ensuring that the frontend application and API endpoints remain stable. This evolution highlights the system's adaptability to new technological demands without disrupting the established flow of operations.

These enhancements reflect a keen awareness of the system's operational needs and a forward-looking approach to scalability and performance. The transition to a timeseries database and the retention of existing API interfaces demonstrate a balance between innovation and stability, crucial for maintaining service quality in a dynamic technological landscape.

# 7. Conclusions

The thesis entitled "Integration of Time Series Databases with Relational Databases for Data Series Management" presents a comprehensive and detailed study, focusing on the comparative performance and applicability of Time Series Databases (TSDBs) and traditional SQL databases, particularly in the context of managing large-scale, time-stamped data. The research primarily revolves around the evaluation of InfluxDB, a prominent TSDB, against MySQL, a widely used SQL database, and includes Apache Druid, another TSDB, for a broader perspective.

The core of the thesis lies in its meticulous performance analysis. It dives deep into various operational aspects such as data ingestion rates, query response times, and overall system efficiency under different database conditions. InfluxDB stands out in this analysis, demonstrating a clear edge in handling large datasets and real-time data processing. The efficiency of InfluxDB is particularly notable in scenarios involving empty databases, databases filled with extensive historical data, and fully occupied tables. This distinction in performance is attributed to the inherent architectural advantages of TSDBs in managing time-centric data, which becomes increasingly relevant in sectors where real-time analytics and quick data processing are critical.

A significant part of the thesis is dedicated to exploring the challenges associated with transitioning from a conventional SQL database system like MySQL to a more specialized TSDB. This transition, as the research indicates, is not without its complexities. The study emphasizes the need to consider numerous factors, such as the reformation of data schema to align with time-series models, the adaptation to new query languages tailored for TSDBs, and the overarching operational changes required in database management practices.

In addressing these challenges, the thesis proposes an innovative hybrid approach, suggesting a time-series database with MySQL. This approach is presented as a solution that capitalizes on the unique strengths of both database systems. By employing a TSDB for tasks that involve extensive time-based data operations such as data cleaning and aggregation, and using MySQL for general-purpose data management, the hybrid approach aims to enhance the overall efficiency and scalability of data processing systems.

The implications of the study are far-reaching, particularly considering the increasing importance of efficient data management in the digital era. The findings advocate for a broader adoption of TSDBs in applications where real-time data analysis is paramount. This recommendation is underpinned by the growing volume and complexity of data in various industries, notably in the energy sector, where managing time-stamped data efficiently is crucial for operational success.

To conclude, the thesis makes a compelling case for the adoption of TSDBs like InfluxDB, especially in scenarios where traditional SQL databases may fall short in terms of performance and scalability. It also provides valuable insights and practical solutions for organizations grappling with the decision to transition between these technologies. The study not only underscores the superior capabilities of Time Series Databases in managing time-based data but also highlights their increasing relevance and potential in the landscape of modern data management.

# 8. Source Code

For those wishing to examine our project in more detail, we provide you with the link to the GitHub repository where the source code of the application for this thesis is available. Link to the repository: https://github.com/ntua/timeseries22

# Bibliography

[1] Engines ranking. https://db-engines.com/en/ranking/time+series+dbms.

[2] S. Di Martino, L. Fiadone, A. Peron, A. Riccabone, and V. N. Vitale. Industrial internet of things: persistence for time series with NoSQL databases. In 2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), pages 340–345. IEEE, 2019.

[3] C. P. District. Beach water quality - automated sensors: City of Chicago: Data portal.

https://data.cityofchicago.org/Parks-Recreation/Beach-Water-Quality-Automated-Sensors/qmqz2xku

[4] A. S. Foundation. Database for modern analytics applications. https://druid.apache.org/

[5] InfluxDB. Open-source time series database. https://www.influxdata.com/.

[6] R. Liu and J. Yuan. Benchmarking time series databases with iotdb-benchmark for IOT scenarios. arXiv preprint arXiv:1901.08304, 2019.

[7] S. N. Z. Naqvi, S. Yfantidou, and E. Zimányi. Time series databases and influxdb. Studienarbeit, Université Libre de Bruxelles, 12, 2017.

[8] TimescaleDB. Timeseries database for PostgreSQL. https://docs.timescale.com/

[9] Bonil Shah, P. M. Jat and Kalyan Sasidhar PERFORMANCE STUDY OF TIME SERIES DATABASES https://arxiv.org/pdf/2208.13982.pdf

[10] Ted Dunning, Ellen Friedman. Time Series Databases: New Ways to Store and Access Data. O'Reilly Media, Inc., December 2014. ISBN: 9781491914724.

[11] InfluxData. "Time Series Database." Accessed November 12, 2023. https://www.influxdata.com/time-series-database/.

[12] InfluxData. "InfluxDB vs Apache Druid." Accessed November 12, 2023. https://www.influxdata.com/comparison/influxdb-vs-druid/.

[13] G2. "Time Series Databases." Accessed November 12, 2023. https://www.g2.com/categories/time-series-databases.

[14] Imply. "Druid Architecture Concepts." Accessed November 12, 2023. https://imply.io/druid-architecture-concepts/.

[15] InfluxData. "InfluxDB 3.0 System Architecture." Accessed November 12, 2023. https://www.influxdata.com/blog/influxdb-3-0-system-architecture/.

[16] B. Leighton, S. J. D. Cox, N. J. Car, M. P. Stenson, J. Vleeshouwer, and J. Hodge, "A Best of Both Worlds Approach to Complex, Efficient, Time Series Data Delivery," in R. Denzer et al. (Eds.): ISESS 2015, IFIP AICT 448, pp. 371–379, 2015

[17] V. Rudakov, M. Timur, and A. Yedilkhan, "Comparison of Time Series Databases," in Proceedings of the 17th International Conference on Electronics Computer and Computation (ICECCO), 2023. DOI: 10.1109/ICECCO58239.2023.10147153

[18] C. Praschl, S. Pritz, O. Krauss, and M. Harrer, "A Comparison of Relational, NoSQL and NewSQL Database Management Systems for the Persistence of Time Series Data," in Proceedings of the International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME), 16-18 November 2022, Maldives. DOI: 10.1109/ICECCME55909.2022.9988333

[19] T. Verner-Carlsson, V. Lomanto, "A Comparative Analysis of Database Management Systems for Time Series Data," in School of Electrical Engineering and Computer Science, 21 June 2023, Stockholm, Sweden. Host company: Zenon AB.

[20] Python 3.12.2 documentation Accessed November 12, 2023. https://docs.python.org/3/library/time.html#time.perf_counter