Data Management for mobile applications dependent on geo-located data

Nuno M. Santos, Luís M. Silva, João Leitão, Nuno Preguiça
NOVA LINCS & NOVA School of Science and Technology
{nmf.santos,lmt.silva}@campus.fct.unl.pt,{jc.leitao,nuno.preguica}@fct.unl.pt

Abstract

An increasing number of mobile applications share location-dependent information, from collaborative applications and social networks to location-based games. In such applications, users are interested in information related to their immediate surroundings or destination when moving instead of data referring to events or state in distant areas. The current database systems enforce uniform consistency models that do not take into consideration data geographical locality, requiring applications to implement ad-hoc solutions that are sub-optimal at best, and can lead to poor performance in the worst case.

In this paper, we argue in favour of consistency models where data location is a key property of data items that is leveraged to govern the operation of replication protocols and the guarantees provided to data accessed by users. To illustrate this, we present FocusDB, a new data management system designed to leverage both object and client location to combine stronger and weaker levels of consistency on a per-object basis. The system discussed here represents a first step in a larger ongoing research effort focused on deriving new consistency models and replication protocols that leverage our previous observation.

Keywords: Location-based Data, Tunable Consistency, Data Replication, Mobile Applications.

ACM Reference Format:
Nuno M. Santos, Luís M. Silva, João Leitão, Nuno Preguiça. 2023. Data Management for mobile applications dependent on geo-located data. In *Proceedings of 10th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC 2023).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Modern distributed applications have increasingly shifted towards supporting the interaction among users, with mobile devices becoming one the primary mediums for deploying and interacting with new applications. Therefore, supporting efficient access to shared data among mobile devices has become more relevant. Examples of such applications include multiplayer games, maps services with real-time crowdsourced traffic information and alerts, or any data distribution system serving localised content such as ads, local news alerts, or social media features.

A factor that is common to all these applications is that data objects have an associated geographic location, and the location of users impacts significantly the data objects that are accessed and modified by those users. This offers the opportunity to design distributed data storage systems that can be optimised by taking into consideration this factor when dealing with replica placement and access pattern prediction. Maybe less obviously, this also offers the opportunity to have specialised data replication protocols that take into consideration the location associated with data objects, since users will be less concerned about the detail or freshness of data that is far away from their current positions (or to their destination, when moving). This allows to focus the operation of a distributed data management systems, and associated replication protocols, in providing the greater detail and improved data freshness for data that is associated to locations in the vicinity of users, which will be of primary interest to them.

However, existing commercial data storage solutions, such as Cassandra [11], MongoDB [21], CosmosDB [22], among others are designed to be general purpose, and therefore do not take into consideration in their data models and replication protocols aspects such as data (and user) location. Conversely, current consistency models, from strong consistency to weak consistency, are not defined to take into consideration data properties such as location to define the guarantees that are provided to users that access replicas (for instance considering the location of that user). Consequently, they cannot adapt their behaviours or provide clear guarantees regarding the location associated with data objects and the location to where data is replicated being it other servers or mobile clients. [23] This hinders the opportunity to further improve the performance and scalability of the current state of the art solutions at best (in terms of latency, throughput, and overall usefulness of data exposed to clients), and can lead to performance degradation and lower user experience at its worse.

To respond to this need, we present FocusDB, a first step in a wider research effort dedicated to propose novel data consistency models and replication protocols that use location associated with data, server replicas, and users as a primary factor to govern the data management strategy. FocusDB achieves this first step through two main contributions. The first is a novel data model that is: *i*) capable of adapting the consistency guarantees enforced on data exposed to clients taking into account the location associated to data and the indications of the user related to locations of interest; and *ii*) automatically adjusts the detail of exposed data, through aggregations, that are guided by the location associated to the data and the user. The second contribution is a client data caching mechanism that leverages the locations of interest provided by the user, to ensure good performance even in low connectivity scenarios. We have developed an initial prototype of FocusDB using MongoDB as the underlying technology for the system storage layer. However, any other type of database, such as geospatial databases, could have been explored since the developed model is agnostic to how the data is stored. Our initial experimental results show that FocusDB, despite not fully realising our vision for geo-located data models, consistency models, and replications protocols, already brings significant advantages in the context of a use case application dedicated to assist users in finding parking spots for cars within cities.

The remainder of this paper is organised as follows. Section 2 discusses relevant related work and background. Section 3 presents the system model and architecture adopted in the design of FocusDB. Section 4 discusses the data model used in FocusDB while Section 5 provides details on the design and implementation of the main components of FocusDB. Section 6 presents the use case application used in the evaluation work reported in Section 7. Finally, Section 8 concludes the paper and discusses future work.

2 Related Work and Background

The section overview related works in distinct areas.

Data Replication: Replicating data is a standard technique in distributed systems used to tolerate failures, ensure high availability, and improve performance. Some systems [3, 13] implement replication solutions that provide strong consistency, creating the illusion of a single copy of the database. Strong consistency is often unsuitable for mobile computing settings where it is useful to have replicas of some objects in mobile devices, which may frequently become disconnected since such consistency models require coordination among replicas to perform operations.

In those contexts, weak consistency solutions, such as eventual consistency [29], are more commonly used, where operations can be performed on a replica without needing immediate coordination with other replicas [4, 16, 25]. Weak consistency solutions often lack restrictions on the amount of inconsistency between replicas, which can be a limiting factor in the development of applications where users need some form of data freshness guarantees. Approaches like the Escrow Transactional Method [15], Epsilon Serializability[18], and Continuous Consistency Model [30] impose limits on the amount of inconsistency a system can tolerate, regulating data operations to prevent from exceeding the established limits without coordination.

The authors of [20] propose a system where the consistency level of a replica can change over time, depending on its distance from a pivot point. Unlike these systems, FocusDB reduces the consistency of the data based on a location property that is intrinsically associated with each individual data object, hence leveraging a novel data model. FogStore [6], shares similarities with FocusDB, particularly regarding the notion of Context of Interest. In FogStore, data requests made within a region of interest are assigned a strong consistency level, while requests outside the region receive a weak consistency level. However, FocusDB differs from FogStore as it not only adapts the consistency level of data but also can reduce the detail of the data exposed to clients (based on aggregations) to achieve improved performance.

Mobile Systems: In the field of mobile computing, several systems have been developed to address the specific challenges of this computing environment [16, 25–28]. For example, Bayou [25], Simba [16], and Legion [26] offer operations with weak consistency guarantees, while Fidelity [28] provides strong consistency but with limited performance in terms of latency and throughput. The system described in [27] allows users to specify their desired consistency level for data access RethinkDB [19] provides real-time push notifications to connected clients, aiming for low latency, high throughput real-time interactions. However, none of the above focus on the location associated with data objects, servers, and clients to govern the system's operation offering different guarantees on data exposed to clients.

Time-series databases: When dealing with large amounts of time-stamped data from sources such as IoT devices, financial transactions, or sensor networks, time-series databases can provide highly efficient solutions. InfluxDB [10] is a distributed time-series database that uses a SQL-like query language and supports data retention policies for data expiration. TimescaleDB [9] is built on top of PostgreSQL and provides automated data chunking and compression for efficient storage and retrieval. Graphite [1] stores data hierarchically and supports various data sources and visualisations. Additionally, Prometheus [17] is designed for monitoring and alerting, supporting data scraping, aggregation, and querying. Time-series databases offer optimised performance for data with a time attribute - in our work we try to achieve similar benefit for data with a location attribute.

Furthermore, FocusDB focuses on adapting the consistency offered to different clients based on interests provided by the user regarding specific locations. This allows applications to exploit better a trade-off between consistency and performance based on application-specific logic.

3 FocusDB Architecture and Overview

FocusDB is a data management and replication system aimed at supporting applications running in mobile devices. With FocusDB, developers can easily define the data model for their application and specify how that data should be organised and delivered to clients. Data has an associated location and can be delivered at different levels of detail. The system architecture consists of services that store and distribute application data to clients according to their preferences.

System Model and Design The system includes multiple mobile clients communicating with a central server over a network. Clients can request data or execute actions, and the server replies accordingly, managing and providing consistent data views to all clients that depend on the client locations of interest.

In FocusDB, all data objects have a location property that identifies the location associated with that data object. When clients create new data objects, these will either be associated with a location indicated by the user or optionally be tagged as the location of the user device when the object is created.

Data objects can have any number of attributes. However, some of these attributes may only be relevant for users at a given proximity of the location associated with that data object. Hence FocusDB is able to filter the attributes of an object that are exposed to a client based on the distance of the user to the location associated with that data object. To this end, all client operations are geographically tagged by the mobile applications with geographic metadata that FocusDB leverages for this purpose. This allows FocusDB to expose multiple representations of a data object as users reach the location associated with that data object.

The Cloud-Side Architecture. The cloud-side architecture is composed by two primary components: an API and storage layers. The API layer is an intermediary between the clients and the database, allowing access to data access and manipulation methods. It is responsible for translating client requests into database queries, executing the logic of requests, and proactively notifying clients about changes relevant to the clients' cached data. The storage layer manages data collections (providing durability) and a set of materialised views based on the data attributes. These views consist of unique data objects or aggregations of objects grouped by a specific attribute to display different levels of detail. This component is supported by a database system. We do not prescribe a specific database system since the FocusDB API layer can be easily adapted for different solutions.

The Client-Side Architecture. FocusDB clients interact with the servers using a client library, submitting queries and updates and receiving change notifications. Clients keep a local cache for supporting immediate replies to applications and to reduce the on the central component.

When the client performs a read operation, it sends a data request to the API, which communicates with the storage layer to retrieve relevant data. The API then returns the data to the client, which stores it in its local cache, and subscribes to notifications for the returned data. The server notifies the client whenever a write operation modifies, adds, or removes data relevant to that request.

Note that depending on the distance of the client to the location of returned data objects, not all updates will generate a notification to the client (i.e., clients in close vicinity will receive more updates than clients farther away from the location of those data objects). For a write operation, the client submits the request to the FocusDB API, which contacts the storage layer to reflect the change in the data layers. If the database system does not have a view manager, FocusDB's view manager is triggered asynchronously to update all the relevant entries in the views being managed by the system.

4 FocusDB Data Model

The data model for our system acts as a key/value store and offers different consistency levels for the same object.

Objects represent individual data elements stored in the system, assigned a unique identifier, and stored in collections with a flexible schema. Data objects have multiple attributes, depending on the application. We name *Base Data*, the objects written directly by the application, and *Derived Data* the objects computed from Base Data or other Derived Data objects. These computations can be as simple as reducing the visibility of specific attributes or as complex as combining features from multiple objects using operations such as union, intersect, and general aggregations. The results are stored in the Derived Data object and can be used for further analysis or operations. The system supports basic CRUD operations on both Base Data and Derived Data objects. Changes made to objects can be carried out by both clients and servers, triggering a view to be recomputed. However, clients cannot change views directly. Instead, they must operate over full representations of Base Data objects.

Collections are system-wide and do not assume object locality. A single collection maintains all the Base Data objects. A node should contain a subset of the collection, containing the objects pertaining to that replica node location. A single collection can store base and derived objects, or multiple collections can be employed, one for each data type.

Views are defined as a function over a data collection or other views. They are built concerning the global data collection, meaning that the definition of views is global within the system. A view can filter data fidelity, exploiting its georeferenced attributes. It is not bounded to any particular data object or locality properties of one client. The definition of a view is materialised in an object. It contains a name, the collection it applies to, and, most importantly, a function representing the query used to compute that view. Views are materialised server-side to avoid overloading clients with frequent computations. The views may need to be updated as Base Data reflected in them are updated. Views have limited geographical relevance, allowing for redirection of computations to the cloud and edge server responsible for that region.

5 FocusDB Components

Data API At the Data API level, two categories of operations can be defined: those to be performed by the programmers for managing collections and views, and the remaining belonging to the client interaction with actual data. The first kind comprises endpoints for views and collections operations as its creation and update. The second includes all read and write operations (inserts, updates, deletes, and gets) for the data objects.

Two parameters must be defined for the data read endpoint: the view to be accessed and a condition. The condition can be of two types. If the condition is a geographical area, the system returns all data objects whose location belongs to that area. On the other hand, if the condition is a query based on a data object attribute, the system returns only the objects from the attribute view that belong to the inputs that correspond to the query. We assume this attribute has geographical meaning, serving as a code for an area, but it is not a requirement.

The view selected in the request also alludes to an implicit choice at the level of data detail. It is determined that the responsibility of increasing or decreasing the level of detail is that of the application using the FocusDB client, which implies that the application must be able to determine when it is appropriate to increase the level of detail of the requests. With the write endpoints, a client can insert new objects and update or remove those already in the database.

The reader should note that in this work, we do not consider operations issued by malicious clients, leaving this point for future iterations.

Incremental View Manager Since read operations are performed on materialised views of the system, these must be appropriately updated so that clients can receive the latest data version. However, only some database systems perform incremental management of materialised views, and those that perform do not take location properties into consideration to govern this process. [2]

In systems lacking such a mechanism, entries of each view are instantiated based on the collection contents at that moment. That is the case of MongoDB, which our prototype uses as the storage backend. As the base collection suffers modifications, the views remain static and do not reflect the current state of the data unless manually requested. Therefore it is necessary to address the lack of this component so that the system can operate more efficiently.

In the context of FocusDB, the incremental view management system consists of a component responsible for reacting to updates on the data collections by reflecting them in the materialised views. This component acts asynchronously concerning the API, as updating the views can be time-consuming. From that, slight inconsistencies can be introduced during the update process. However, given the reduced dimension of the inconsistency and the benefits of the component, we can consider that acceptable.

Materialised views are pre-computed and stored in collections, speeding up query processing by avoiding recomputing the results of a query each time it is executed. When an object is created or modified, the incremental view manager processes the modifications and determines the impact on the materialised views. To improve the overall process operations are queued per view. Only after a certain threshold will the view manager batch all the operations concerning each view, followed by the batch execution of the updates. If the update affects a materialised view, the incremental view manager recomputes only the portion of the view that needs to be updated and stores the revised results in the database. This approach is more efficient than recomputing the entire view, as it only updates the portion that needs to be changed.

Notification System This component ensures that client-side cached data does not become outdated in relation to the data maintained in the centralised component of FocusDB. When a client requests data through the API, in most cases, it declares that it is interested in a specific data set and therefore wants the information belonging to that set to be kept updated over time.

We could encumber the client by determining that the mobile application is responsible for periodically refreshing its cached relevant objects by reissuing the requests that created those cache entries However, this method would be inefficient since the application itself has no notion of whether the data that the user is interested in has been changed or not. In cases where data is rarely updated, such a pooling approach would result in unnecessary resource consumption both for request processing and communication.

To avoid this, FocusDB relies on a notification system that is responsible for notifying clients when relevant updates happen to data that is cached by the client and whose the client has registered interest. This mechanism works as follows: when a client accesses data (through the API) for the first time (being it data for a location never accessed or accessing data previously accessed by with a different detail level) the client can request also to become subscribed to the information channel associated with that data (which is achieved by activating a flag on the data requests). If the client does so, the system subscribes the client to the interest set corresponding to the data returned by that request. From that moment on, any updates that changes the data returned by that request are also propagated to all subscribers of that interest set. This prevents unnecessary data requests from mobile applications.

This mechanism is relevant towards our approach of location-dependent consistency guarantees. Note that data that refers to locations further away from the user locations of interest might less relevant to the user. In those cases, this mechanism, allows to avoid transmitting all individual updates for that data, or making them with a lower frequency, than for data that is highly relevant for the user, allowing to reduce traffic even further.

6 Use Case Application

FocusDB unique approach makes it suitable for various application scenarios, such as multiplayer games, map services, and hyperlocal data distribution systems. For instance, FocusDB can provide real-time data sharing based on the location of users in multiplayer interactive games [8, 12], players can subscribe to data near their location, and the system can adapt the consistency and exposed detail of the data based on locations to ensure optimal performance. Filter and deliver relevant information to users in mapping services [7] by implementing real-time crowdsourced traffic and alert information filtered based on the current location of the used. Adapt the level of detail of the data based on device location in hyperlocal data distribution systems [5, 24], saving bandwidth for the client and costs for advertisers.

In this paper we focus on a use case location-based application that showcases the need for a new consistency model.

Where to Park?

For many drivers in densely populated areas, finding a parking spot is a daily challenge. Whether commuting to work or visiting an unfamiliar place, knowing where to park can make all the difference. That is the motivation for our envisioned use case, the *Where to Park?* application.

The *Where to Park?* application is a mobile application for smartphones and automobiles that keeps track of and helps find empty parking spots. The app provides users with multiple levels of detail on parking availability across in different regions, from city-wide to street-level views. Users can see an approximate number of available spots for an area or zoom in on the street and see all the parking locations and their availability. The app can be used for both street parking (free or paid) and private parking facilities (e.g. shopping malls). For private parking, the app can retrieve real-time data on availability through an API provided by the parking operator. For street parking, the app can rely on a network of sensors installed in each spot that communicates with nearby parking meters or crowdsourced data from users who report available spots.

7 Evaluation

We conducted a series of experiments to evaluate the performance of FocusDB in a mobile application scenario, relying on the use case application introduced in the previous section. Our system, called FocusDB (labelled *FOCUS* on plots), is equipped with Incremental View Manager and Notification System mechanisms and is backed by a MongoDB database. To evaluate the system, we built a workload generator that mimics multiple mobile clients. This workload generator can launch a configurable number of clients, each running a set of predefined trips and workloads, to realistically evaluate FocusDB's capabilities with a realistic usage. To enhance the test environment and introduce an element of unpredictability, we implemented a server-side Ghost Client mechanism that randomly changes the status of chosen parking spots, keeping the dataset dynamic and invalidating cached data. We also implemented two additional ways the application could interact with data to compare our implementation. The first method, (dubbed *BASIC* in plots), involves translating the client's request directly to MongoDB Queries, which the server executed directly on the database, providing to clients full object representations. The second method, (labelled *INDEX*) in plots, allows the MongoDB instance to create indexes on relevant attributes that the client application would query, speeding up server responses.

Geographic dataset. We have leveraged real datasets to generate the movements of clients and changes to data objects in our experiments. The first dataset includes the geographical coordinates of roughly 500 taxi trips in the city of Porto, Portugal, and was enriched with the assistance of the OpenRouteService API [14]. This resulted in a dataset with more precise coordinates per trip. The second dataset contains the coordinates of approximately 500, 000 parking locations in Porto, derived from the taxi trips dataset, to ensure that the trips executed by clients intersect with parking spots. We used the taxi trips dataset to compute client movements and the parking spots dataset to represent the locations of relevant data objects.

Experimental setup. In our experiments, we used the Nova SST Computer Science Department's Compute Cluster, which offers a variety of machine types. Both the server-side and clients were deployed on machines equipped with an AMD EPYC 7281 CPU (16 cores/32 threads), 128 GiB DDR4 2666 MHz of RAM, and dual 10 Gbps network capacity. On the server machine, we deployed an instance of FocusDB, supported by a MongoDB database that was seeded with the 500, 000 parking spots, precomputed views at the neighbourhood and street level, and a RabbitMQ instance to support the notification system. Use used a single instance of the workload generator that was responsible for managing the lifecycle of multiple simulated a variable number mobile clients. Each simulated client makes requests to the server based on a workload derived from the taxi trips dataset.

Experiment scenario. Our experiments focus on comparing the performance of FocusDB with the two alternative architectures (*BASIC* and *INDEX*) when the requests issued by clients are limited to read operations. With using FocusDB,

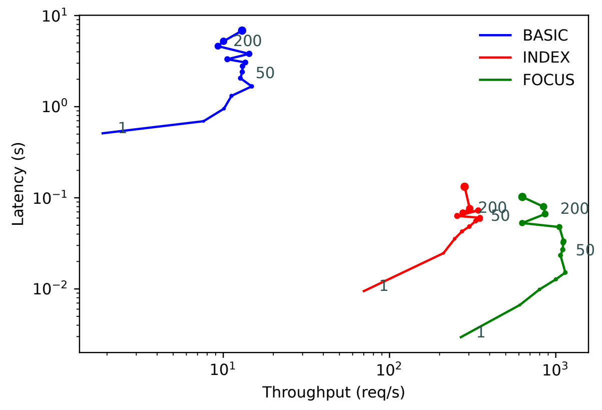

Figure 1. Throughput Latency per Number of Clients

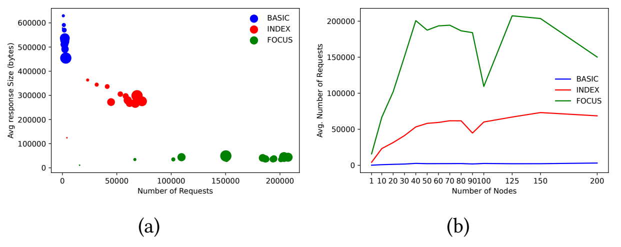

Figure 2. (a) Avg. Response Size per Avg. Requests and (b) Avg. Number of Requests per Number of Clients

the client initiates a request by specifying the end location of the trip, which indicates the interest set regarding parking spots. In response, the client receives a neighbourhood view with a count of available parking spots in that area. As the client approaches the destination, the level of detail in the exposed data increases, with the client obtaining a count of free spots on the desired street. When the client is almost at the destination, the level of detail increases again, and the requests return complete objects describing the parking spots on the street, enabling the client to choose a spot to park. During each experiment, the request detail distribution for each trip is set at 80% aggregations and 20% full objects. Each client performs five trips per test, which are repeated three times, and the results are averaged. Ghost Clients simulate 100 parking spots availability changes every two seconds. These settings are applied for the Basic and Index implementations, with all requests producing full objects. The experiment measures throughput and latency, evaluated as a function of the number of clients and hence the total number of client requests.

Figure 1 presents a throughput × latency plots comparing all alternatives, thatshows that using FocusDB (*FOCUS*) leads to improved performance when compared with the *BASIC* and *INDEX* alternatives. The plot shows that the mechanisms employed in the design of FocusDB allows to support similar number of clients offering lower latency and achieving a higher throughput. The *BASIC* alternative exhibits the lowest throughput and highest latency among the three alternatives. The *INDEX* alternative falls in between the other two in terms of performance.

Figure 2 captures some of the factors that explains the performance improvement of FocusDB, which are derived from leveraging the location associated with data objects and clients issuing requests. Figure 2(a) reports on the size of the responses issued by the centralised component to clients. It clearly shows that the response size of FocusDB is significantly smaller when compared with the two other alternatives. This leads to a more effective usage of network resources (and computational resources to compute those answers). Figure 2(b) reports the number of requests issued by clients of the different alternatives for an increasing number of clients. It shows that the caching mechanism of FocusDB combined with its notification system, allows for an extreme reduction on the requests issued to the centralised component. Note that our notification mechanism allows clients to still have access to up-to-date date for the locations of relevance for the client.

8 Conclusion and Future Work

In this work, we presented FocusDB, a novel data management system specifically designed to support geo-located data in a mobile environment. The system's two main components, a flexible data model and a client caching mechanisms and notification system, work together to adapt data delivery based on both location associated with data objects and the location of the user. Our preliminary experimental evaluation shows that FocusDB, when compared with classical alternatives, shows significant performance improvements by leveraging our data model that allows to expose different levels of detail based on location.

This effort is the first on a longer research line that aims at exploiting the location property as a governing factor to build distributed large-scale data stores. In particular we plan on formalising consistency models that take into account location of both data objects, and servers/clients manipulating that data, and efficient replication protocols that can implement such consistency models.

Acknowledgments

This work was partially supported by FCT/MCTES grants, through NOVA LINCS (UIDB/04516/2020) and the PhD Research Scholarship awarded to Luís Silva (2021.05686.BD). Dataset expansion was possible by API access to OpenRouteService from HeiGIT gGmbH.

References

[1] 2023. Graphite DB. http://graphiteapp.org/
[2] Abdulaziz S Almazyad and Mohammad Khubeb Siddiqui. 2010. Incremental view maintenance: an algorithmic approach. *International Journal of Electrical & Computer Sciences* 10 (2010).
[3] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev,

Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[4] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.

[5] Foursquare. 2023. Foursquare City Guide. https://foursquare.com/city-guide

[6] Harshit Gupta and Umakishore Ramachandran. 2018. Fogstore: A geo-distributed key-value store guaranteeing low latency for strongly consistent access. In *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*. 148–159.

[7] Google Inc. 2023. Nearby API. https://developers.google.com/nearby

[8] Niantic Inc. 2023. Pokémon GO. https://pokemongolive.com/

[9] Timescale Inc. 2023. TimescaleDB : SQL made scalable for time-series data. *TimescaleDB Whitepaper*.

[10] InfluxData. 2023. InfluxDB Times Series Data Platform | InfluxData. https://www.influxdata.com/

[11] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review* 44, 2 (2010), 35–40.

[12] João Leitão, Pedro Ákos Costa, Maria Cecília Gomes, and Nuno M. Preguiça. 2018. Towards Enabling Novel Edge-Enabled Applications. *CoRR* abs/1805.06989 (2018). arXiv:1805.06989 http://arxiv.org/abs/1805.06989

[13] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. 2013. Low-Latency Multi-Datacenter Databases Using Replicated Commit. *Proc. VLDB Endow.* 6, 9 (jul 2013), 661–672. https://doi.org/10.14778/2536360.2536366

[14] Pascal Neis and Alexander Zipf. 2008. Openrouteservice. org is three times "open": Combining OpenSource, OpenLS and OpenStreetMaps. *GIS Research UK (GISRUK 08). Manchester* (2008).

[15] Patrick E O'Neil. 1986. The escrow transactional method. *ACM Transactions on Database Systems (TODS)* 11, 4 (1986), 405–430.

[16] Dorian Perkins, Nitin Agrawal, Akshat Aranya, Curtis Yu, Younghwan Go, Harsha V Madhyastha, and Cristian Ungureanu. 2015. Simba: Tunable end-to-end data consistency for mobile apps. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–16.

[17] Björn Rabenstein and Julius Volz. 2015. Prometheus: A Next-Generation Monitoring System (Talk). USENIX Association, Dublin.

[18] Krithi Ramamritham and Calton Pu. 1995. A formal characterization of epsilon serializability. *IEEE Transactions on Knowledge and Data Engineering* 7, 6 (1995), 997–1007.

[19] RethinkDB. [n.d.]. RethinkDB Architecture. https://rethinkdb.com/docs/architecture/

[20] Nuno Santos, Luís Veiga, and Paulo Ferreira. 2007. Vector-field consistency for ad-hoc gaming. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 80–100.

[21] William Schultz, Tess Avitabile, and Alyson Cabral. 2019. Tunable consistency in mongodb. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2071–2081.

[22] Dharma Shukla, Shireesh Thota, Karthik Raman, Madhan Gajendran, Ankur Shah, Sergii Ziuzin, Krishnan Sundaram, Miguel Gonzalez Guajardo, Anna Wawrzyniak, Samer Boshra, et al. 2015. Schema-agnostic indexing with Azure DocumentDB. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1668–1679.

[23] Luís M Silva, Frederico Aleixo, Albert van der Linde, João Leitão, and Nuno Preguiça. 2022. Geo-located data for better dynamic replication. *arXiv preprint arXiv:2205.01045* (2022).

[24] Telegram Team. 2019. Location-Based Chats, Adding Contacts Without Phone Numbers and More. https://telegram.org/blog/contacts-local-groups

[25] Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer, and Carl H Hauser. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. *ACM SIGOPS Operating Systems Review* 29, 5 (1995), 172–182.

[26] Albert van der Linde, Pedro Fouto, João Leitão, Nuno Preguiça, Santiago Castiñeira, and Annette Bieniusa. 2017. Legion: Enriching internet services with peer-to-peer interactions. In *Proceedings of the 26th International Conference on World Wide Web*. 283–292.

[27] Albert van der Linde, João Leitão, and Nuno Preguiça. 2020. Practical client-side replication: weak consistency semantics for insecure settings. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2590–2605.

[28] Kaushik Veeraraghavan, Venugopalan Ramasubramanian, Thomas L Rodeheffer, Douglas B Terry, and Ted Wobber. 2009. Fidelity-aware replication for mobile devices. In *Proceedings of the 7th international conference on Mobile systems, applications, and services*. 83–94.

[29] Werner Vogels. 2009. Eventually Consistent. *Commun. ACM* 52, 1 (jan 2009), 40–44. https://doi.org/10.1145/1435417.1435432

[30] Haifeng Yu. 2000. Design and evaluation of a continuous consistency model for replicated services. In *Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*.