

# Adaptive Caching of Spatial Queries in PostGIS Using H3-Based Indexing

Mohammed Shakir

`mohsha-0@student.ltu.se`

Department of Computer Science, Electrical and Space Engineering



January 19, 2026

# Abstract

x

## Acknowledgements

I would like to thank my internal supervisor and examiner at Luleå University of Technology (LTU), Tatiana Liakh, for steady guidance, clear feedback, and for helping me keep the scope realistic. I am especially grateful to my external supervisor at Softhouse, Axel Lassinantti, whose advice, reviews, and day-to-day help shaped the project; this thesis would not have come together without his support.

I also thank Metria AB for hosting the work and providing access to their platform. In particular, I am grateful to Erik Wikström at Metria for preparing datasets, answering many practical questions, and reviewing intermediate results. Softhouse deserves thanks for connecting me with Metria and for offering a great workspace during the project.

Finally, I want to thank my classmates and friends, Elliot Wallin and Suad Huseynli, for carefully reading drafts and giving honest feedback before submission.

# Contents

<b>Abstract</b>	<b>2</b>
<b>Acknowledgements</b>	<b>3</b>
<b>Abbreviations &amp; Acronyms</b>	<b>6</b>
<b>1 Introduction</b>	<b>7</b>
1.1 Background . . . . .	7
1.2 Motivation . . . . .	7
1.3 Problem Statement . . . . .	8
1.4 Research Questions . . . . .	8
1.5 Scope . . . . .	8
1.6 Contributions . . . . .	8
1.7 Thesis Structure . . . . .	9
<b>2 Related Work</b>	<b>10</b>
2.1 Hierarchical spatial indexing and pre-aggregation . . . . .	10
2.2 Adaptive caching for spatial workloads . . . . .	10
2.3 Cache consistency and spatially scoped invalidation . . . . .	11
2.4 Hexagonal grids and H3 . . . . .	11
2.5 Gap analysis . . . . .	11
<b>3 Theory</b>	<b>12</b>
3.1 Conceptual Frame . . . . .	12
3.1.1 Query Models and Spatial Footprints . . . . .	12
3.1.2 Hotspots and Workload Skew . . . . .	12
3.2 Spatial Indexing and Partitioning . . . . .	13
3.2.1 Spatial Indexes in PostGIS . . . . .	13
3.2.2 Grid-Based Bucketing vs. Native Indexing . . . . .	13
3.2.3 Hexagonal Grids and H3 Basics . . . . .	13
3.2.4 H3 Resolutions and Cell Geometry . . . . .	14
3.2.5 Mapping BBoxes/Polygons to Cells . . . . .	14
3.2.6 Spatial Approximation Limits of H3 . . . . .	14
3.3 Caching Theory for Spatial Workloads . . . . .	15
3.3.1 Caching Objectives and Cost Model . . . . .	15
3.3.2 Key Design for Spatial Queries . . . . .	15
3.3.3 Admission and Replacement . . . . .	16
3.3.4 TTL-Based Freshness Control . . . . .	16
3.3.5 Event-Driven Invalidation . . . . .	16
3.4 Consistency and Freshness Semantics . . . . .	17
3.4.1 Definitions: Consistency, Coherence, and Staleness . . . . .	17
3.4.2 Spatial Validity Scopes . . . . .	17
3.4.3 Combining TTL with Event-Driven Updates . . . . .	17
3.4.4 Delivery Semantics and Ordering . . . . .	18
3.5 Performance and Latency Modeling . . . . .	18
3.5.1 End-to-End Latency Decomposition . . . . .	18
3.5.2 Queueing Basics and Little’s Law . . . . .	19

3.5.3	Tail Latency and Percentiles . . . . .	19
3.5.4	Hit Ratio vs. Latency/Throughput Trade-off . . . . .	19
3.5.5	Granularity vs. Memory Cost (H3 Resolution) . . . . .	20
3.5.6	Workload Skew and Hotspot Dynamics . . . . .	20
3.5.7	Golden Signals and Cache Metrics . . . . .	21
<b>4</b>	<b>Method</b>	<b>22</b>
4.1	Research Approach and Study Design . . . . .	22
4.2	System Under Study and Assumptions . . . . .	22
4.3	Middleware Design and Implementation . . . . .	22
4.3.1	End-to-End Request Flow and Component Responsibilities . . . . .	23
4.3.2	H3 Footprint Mapping and Request Bucketing . . . . .	23
4.3.3	Caching in Redis: Key Schema and Result Storage . . . . .	23
4.3.4	Response Composition and Correctness Safeguards . . . . .	24
4.4	Freshness Mechanisms . . . . .	24
4.4.1	TTL Policy and Configuration . . . . .	24
4.4.2	Kafka/CDC Event Processing and Spatial Invalidation . . . . .	24
4.5	Experimental Methodology . . . . .	25
4.5.1	Testbed and Deployment (Containers, Versions, Hardware) . . . . .	25
4.5.2	Baselines and Compared Configurations . . . . .	25
4.5.3	Experiment Procedure . . . . .	26
<b>5</b>	<b>Results</b>	<b>27</b>
<b>6</b>	<b>Discussion</b>	<b>28</b>
<b>7</b>	<b>Conclusions</b>	<b>29</b>
	<b>References</b>	<b>30</b>

## Abbreviations & Acronyms

x

# 1 Introduction

## 1.1 Background

Operational web mapping stacks at geospatial service providers typically combine GeoServer for OGC-compliant services with a PostGIS spatial database. Raster map tiles can be served via WMS/WMTS endpoints, which predefined tile matrices to improve the delivery of rendered images [27]. In contrast, vector feature access through WFS exposes fine-grained queries over feature properties and geometries, enabling filtering, editing, and analytics workflows that do not align cleanly with fixed tile boundaries [26]. On the database side, PostGIS provides spatial types and functions with GiST-based indexes that are important for performance [31]; yet when traffic is highly skewed toward a few urban “hotspot” regions, many requests repeatedly overlap in space and time, re-triggering similar queries and I/O.

Conventional server-side caching (e.g., tile caches) mostly targets rasterized responses; it is less effective for vector queries with complex geometries, attribute filters, or dynamic layers. A better approach is to partition the query footprint into a spatial grid and reuse results for frequently accessed cells [10]. Hexagonal hierarchical grids such as H3 provide near-uniform cell areas across resolutions, consistent neighbors, and identifiers that are convenient as cache keys and for aggregation across multiple zoom levels.

A spatial cache must also stay fresh because upstream data changes. Time-to-live (TTL) expiration gives a simple baseline, but event-driven invalidation, common in change-data-capture (CDC) pipelines built around streaming platforms, can invalidate only the affected spatial cells when features are updated. Combining TTL with event-driven invalidation improves freshness without discarding useful cache entries. Finally, an in-memory store such as Redis provides the fast access needed to reduce end-to-end latency on cache hits while keeping implementation complexity manageable within a middleware layer. This thesis builds on these observations to investigate an H3-keyed, Redis-backed cache between GeoServer and PostGIS, coupled with TTL and Kafka-driven invalidation to handle freshness under skewed spatial workloads.

## 1.2 Motivation

Workloads in operational web mapping services frequently show spatial skew: a small set of geographic areas attracts the majority of traffic, while large regions are queried infrequently. Recomputing overlapping vector queries in such hotspots increases latency and worsens database load even when spatial indexes are present. An H3-guided middleware cache provides three benefits:

1. Lower latency by serving popular regions from memory.
2. Lower PostGIS CPU and I/O by avoiding redundant evaluations.
3. Tunable trade-offs between hit ratio, memory footprint, and freshness by adjusting H3 resolution and invalidation policy.

From a research perspective, this work bridges ideas from hierarchical spatial indexing, adaptive caching, and spatially scoped invalidation into a deployable component that complements rather than replaces existing OGC services. The outcome should be practical recommendations on when adaptive caching is worthwhile, which resolutions to use, how to size memory, and how to integrate event-driven invalidation. The study provides an end-to-end evaluation of H3-keyed caching for dynamic vector workloads, a setting that is less well served by conventional tile caches and has few publicly documented, reproducible benchmarks.

### 1.3 Problem Statement

Modern spatial data platforms serve diverse read workloads through GeoServer and PostGIS. In practice, user traffic is highly skewed; a small set of geographic areas (e.g., city centers) receives a disproportionate share of requests, while large regions are rarely accessed [28]. Under this skew, repeated evaluation of overlapping spatial queries increases latency, drives unnecessary CPU and I/O load in PostGIS, and may slow down downstream map rendering. Conventional tile-oriented caching helps for pre-rendered map tiles [27], but is less effective for dynamic vector queries, highly detailed filters [26], and requests that do not align with fixed tile boundaries [47]. As a result, the platform needs a cache that adapts to *where* and *how often* users query, while remaining correct and fresh when data changes.

The problem addressed in this thesis is to design, implement, and evaluate an *adaptive, spatially aware* caching layer that sits between GeoServer and PostGIS and uses an H3 hexagonal grid to detect hotspot regions from live query traffic, cache corresponding query results in Redis for low latency reuse, and keep cache entries acceptably fresh using a combination of time-to-live (TTL) and event-driven invalidation from Kafka. The solution must reduce end-to-end query latency and database load under skewed workloads without introducing unacceptable staleness or excessive memory overhead. The study aims not only to derive actionable guidance on grid resolution, cache sizing, and invalidation policy, but also to deliver a prototype middleware that can be deployed in real production environments.

### 1.4 Research Questions

To evaluate the proposed approach, this thesis investigates the following research questions:

1. **Effectiveness:** To what extent does H3-based adaptive caching reduce latency and offload query pressure from PostGIS compared to no caching or a simple geometry/TTL cache baseline under skewed workloads?
2. **Granularity:** Which H3 resolution(s) give the best balance between cache hit ratio, memory footprint, and data freshness, given uneven access to different areas?
3. **Invalidation Policies:** What combination of time-to-live (TTL) and event-driven invalidation (triggered by upstream data change notifications) provides acceptable staleness bounds while keeping high performance?

### 1.5 Scope

The thesis develops and evaluates a middleware layer between GeoServer and PostGIS that encodes request footprints as H3 cells, caches vector query results in Redis, tracks hotness, and applies TTL and Kafka-based invalidation. This includes building a working prototype that intercepts spatial requests and serves frequently accessed regions directly from Redis. Experiments will use containerized components and synthetic, skewed workloads representative of urban hotspots. The evaluation focuses on latency percentiles, database load, hit/miss ratios, staleness bounds, and cache memory usage across H3 resolutions, with an emphasis on quantifying the trade-offs among H3 resolution, cache size, hit ratio, and freshness.

### 1.6 Contributions

x



## 1.7 Thesis Structure

The thesis is organised as follows:

- **Introduction** outlines the background, motivation, problem statement, research questions, scope, contributions, and thesis structure.
- **Related Work** surveys prior research on spatial indexing, adaptive caching, and cache consistency for dynamic geodata.
- **Theory** introduces the theoretical background needed to understand H3 indexing, caching principles, and consistency mechanisms.
- **Method** describes the H3-based request bucketing, hotness tracking, cache invalidation policies, and the middleware design.
- **Results** present the experimental setup and report findings across baselines and different H3 resolutions.
- **Discussion** reflects on the results, limitations, and threats to validity and includes considerations of sustainability.
- **Conclusions** summarize the contributions of the thesis and outline directions for future improvements and deployment.

## 2 Related Work

This section frames the thesis in four strands of prior work:

- Hierarchical spatial indexing and pre-aggregation.
- Adaptive caching for spatial workloads.
- Cache consistency and invalidation with spatial scope.
- Hexagonal grid indexing (H3).

The goal is to motivate the chosen H3-guided middleware design and clarify how it differs from existing systems.

### 2.1 Hierarchical spatial indexing and pre-aggregation

A central idea behind the proposed middleware is to divide query footprints into grid cells so that repeated queries over similar regions can be identified and reused. Work on *GeoBlocks* pre-aggregates point data into fine-grained cells and approximates query polygons by unions of those cells. The approximation error is bounded by the chosen cell size, and a trie-like cache reuses previous results for frequently queried regions, adapting to skew over time [47]. Similarly, *Adaptive Cell Trie (ACT)* speeds up point-in-polygon joins by combining multiresolution (quadtree-style) approximations with a radix-tree over cell identifiers; the design avoids most expensive geometric tests while offering user controlled precision [15]. Earlier systems such as *Nanocubes* and the *aR-tree* introduced in-memory spatio-temporal aggregation over hierarchical rectangles; they offer interactive performance but rely on rectangular partitions that can cause unbounded error for arbitrary polygons unless cell size is carefully managed [16, 29]. Together, these works support a grid-based approach that supports multiresolution refinement near boundaries, exposes compact cell identifiers for fast lookup and reuse, and adapts well under workload skew properties leveraged with H3-based bucketing.

### 2.2 Adaptive caching for spatial workloads

Beyond indexing, several systems adapt caching to hotspot access patterns. *STASH* is a distributed, in-memory middleware cache for hierarchical aggregations that chooses replicas based on query frequency and freshness, leading to large latency reductions and a higher query processing rate under skew [21]. *GeoBlocks* cache also adapts automatically as regions get popular [47]. At the opposite end of the stack, *SAC* (Semantic Adaptive Caching) predicts future query windows on the client using partitioned spatial history and prefetches answers to improve responsiveness [17]. Finally, *Vecstra* proposes an OGC-compliant in-memory geospatial cache that integrates caching with standard WFS/WCS interfaces, which is useful when the server side must remain standards based, as in GeoServer/PostGIS deployments [46]. Although not spatial per se, *MinMaxCache* shows how an adaptive in-memory cache can trade the level of aggregation for interactive visualization latency with error, an idea that inspires the evaluation of H3 resolution choices and composition overheads in this thesis [19]. The common thread is that caching should be driven by query semantics, popularity, and acceptable approximation, rather than being a static tile store; the middleware follows this line by caching vector results keyed by H3 cells and adapting to observed hotness.

## 2.3 Cache consistency and spatially scoped invalidation

Keeping cached results fresh is not easy when data and query scopes are spatial. Classic mobile computing studies introduced *location dependent* invalidation: cached answers can become stale not only because the data change but also because the user moves outside a valid geographic scope [48]. This led to approaches like *Bit-Vector with Compression and Implicit Scope Information*, which link cached items directly to their spatial validity regions [48]. Related work on *location dependent semantic caching* also organizes cache segments with explicit spatial scopes and uses them in replacement and validation [18]. The practical takeaway for a server-side cache is to tie each cached entry to an explicit spatial footprint and to invalidate it on time-based expiry (TTL) and spatial overlap with changed features. The proposed design does exactly this: H3 cell keys define validity scopes, while TTL and Kafka-driven events provide temporal and event based invalidation hooks.

## 2.4 Hexagonal grids and H3

H3 is a hierarchical, global grid of hexagonal cells. Its properties include near-uniform area per resolution, consistent adjacency, and a clean hierarchical relation between resolutions that make it attractive for bucketing and composing spatial answers [44]. Empirical reports outside pure research also suggest that hexagons reduce visual and aggregation artifacts compared to rectangular geohashes for many analytics tasks [9]. For our use case, H3 gives a compact key that is independent of the programming language (suitable for Redis), deterministic up/down-sampling across resolutions for composition, and a natural way to map data change events to affected cells for invalidation.

## 2.5 Gap analysis

To the best of current knowledge, none of the above combines, in a single end-to-end system, real-time, *schema agnostic* hotspot detection external to the database via H3 bucketing over arbitrary WFS/WMS requests, *adaptive* caching of vector query results at the middleware layer with measured trade-offs across grid resolutions, and *event-driven* freshness via streaming updates (Kafka) that target exactly the overlapping H3 cells. *Index and cache* systems either pre-compute specific analytics (GeoBlocks, STASH), focus on client-centric prediction (SAC), or discuss standards-compliant caching without fine-grained spatial invalidation (Vecstra). This thesis fills that gap by implementing and evaluating an H3-keyed Redis cache between GeoServer and PostGIS, and by quantifying how resolution, hit ratio, memory footprint, and invalidation policy interact under skewed workloads.

## 3 Theory

This section introduces the concepts that the middleware relies on. This includes how vector queries are expressed and scoped, why real workloads concentrate on a few ‘hot’ areas, and how spatial indexes in PostGIS speed up queries yet still leave room for additional caching. The aim is to establish consistent terminology and assumptions so that later design and evaluation choices (e.g., H3 cell resolution, cache keys, and freshness policies) can be clearly understood and compared.

### 3.1 Conceptual Frame

#### 3.1.1 Query Models and Spatial Footprints

Vector web services define two main types of query filters: a *spatial filter* over geometries and an optional *attribute filter* over feature properties. In OGC-style services, the spatial part ranges from coarse bounding boxes (BBOX) to polygonal predicates such as `Intersects`, `Within`, or `Contains`, while the attribute part uses comparison and logical operators (e.g., `POP_DENSITY > 5000 AND TYPE = 'residential'`). Filter Encoding (FES) defines how these spatial, comparison, and logical operators are combined into a single query expression, in either XML or ECQL form [25].

The term spatial footprint refers to the minimal geometry defining the spatial filter sent by the client (e.g., the BBOX or a user-drawn polygon). For caching theory, the footprint is mapped to a distinct set of spatial buckets (H3 cells) that act as reusable containers. A *spatial response* is the set (or stream) of features matching both filters within the query area, together with any server-side transformations. Reuse then follows two patterns:

- Full reuse: When a new request’s cell set and attribute filter exactly match a cached entry.
- Composed reuse: When the new request spans multiple cached cell entries whose combination can be merged and then filtered.

In both cases, the cache key must encode (layer, cell or cell-set, filter hash, SRID) to preserve identical query meaning with the database result. Freshness is controlled independently via TTL or event-driven invalidation (discussed later), but the scope of what to invalidate remains the same, which is the cells touched by the updated features.

#### 3.1.2 Hotspots and Workload Skew

Interactive map traffic is rarely uniform across space; request popularity typically follows a heavy-tailed (Zipf-like) distribution, where a small number of locations attract a large fraction of requests, while most locations are cold. This pattern is common in many networked systems and motivates cache designs that focus on retaining popular items. In spatial systems, the same pattern appears as *spatial skew*: urban centres, transport corridors, and touristic areas become hotspots, and the set of hotspots may evolve over time with events and daily cycles [28].

Skew has two practical consequences. First, repeated evaluation of overlapping queries in hotspots can dominate query latency and backend load, even when individual queries are optimized with spatial indexes [47]. Second, any effective cache should be *spatially selective*, meaning that it should spend memory on the small subset of cells where reuse is high, rather than caching uniformly across the entire map. Dividing footprints into H3 cells makes it possible to track per-cell popularity, admit only “hot” cells, and combine results for larger queries without caching redundant geometry everywhere.

## 3.2 Spatial Indexing and Partitioning

### 3.2.1 Spatial Indexes in PostGIS

PostGIS optimizes spatial queries using GiST-based indexes that implement an R-tree-like search over geometry bounding boxes. The index performs a fast filter step (bounding-box test) to filter out candidates, followed by an exact geometry check (`ST_Intersects`, `ST_Within`, etc.) on the survivors [31]. Correct use therefore requires both creating the index with `USING GIST` and using index-aware filters so the planner can apply the filter step.

While GiST dramatically reduces the cost of single queries and spatial joins, it does not take advantage of *temporal locality* or *overlap reuse*. If many users repeatedly query the same small region, the database still performs planning, index descent, and exact checks for each request; buffers, caches, and prepared geometries help, but they are not keyed by *where* reuse occurs. An external, cell-keyed cache complements GiST by avoiding the entire evaluation pipeline for popular regions. Once a cell’s result is computed, later requests that map to the same (cell, filter) can be answered from memory. In short, GiST reduces the work needed to find candidates, while spatial caching reduces how often the database has to perform that work at all.

### 3.2.2 Grid-Based Bucketing vs. Native Indexing

Native database indexing (e.g., GiST-backed R-trees in PostGIS) speeds up single queries by filtering out candidates with a fast bounding-box filter and then applying exact geometry filters to the survivors [31]. This *two-pass* evaluation is highly effective per request, but it does not directly capture *temporal locality* across requests that repeatedly target the same small area: the planner, index descent, and exact checks still run for every query.

An external, grid-based bucketing strategy divides space into fixed cells and uses those cells as cache keys for query answers. In this model, a request’s footprint (BBOX or polygon) is mapped to a set of cells; if the (layer, cell-set, filter) tuple is already cached, the middleware composes the response from memory, bypassing the database. Overlapping requests can then reuse previous results because they share cells [47]. The database remains responsible for correctness and misses, while the middleware focuses on detecting and serving hotspots. In short, native indexing reduces the amount of work within a query, while grid bucketing reduces how often the database must do that work at all, by enabling answer reuse across partially overlapping requests. The two techniques complement each other: GiST handles precise geometry tests and complex joins, while grid bucketing leverages spatial clustering to reduce repeated work in hotspot regions.

### 3.2.3 Hexagonal Grids and H3 Basics

H3 is a hierarchical discrete global grid that represents locations as 64-bit cell identifiers [11]. The grid is organized in resolutions ranging from 0 to 15, where each cell has exactly one parent at any lower resolution and up to seven children at the next higher resolution; parent/child navigation is provided by API functions such as `cellToParent` and `cellToChildren` [36] [5]. Adjacency is based on shared edges; neighborhood and vicinity queries use grid traversal operations (e.g., `kRing`, `hexRange`) to list cells within a given graph distance [38].

At resolution 0, the grid contains 122 base cells (derived from an icosahedral projection); refinements add hexagonal children, with pentagons appearing to maintain topology [41]. These properties (stable adjacency, clean up/down sampling, and compact integer keys) make H3 a practical foundation for grouping requests and assembling answers across neighboring cells in a middleware cache.

### 3.2.4 H3 Resolutions and Cell Geometry

Cell size decreases exponentially with resolution. Average hexagon areas are on the order of  $5.16 \text{ km}^2$  at res 7,  $0.737 \text{ km}^2$  at res 8, and  $0.105 \text{ km}^2$  at res 9 (These figures represent global averages, though the exact area depends on each cell’s position relative to the icosahedral faces) [39]. The minimum/maximum area ranges per resolution reflect distortion near icosahedron vertices and the presence of pentagons.

These values give practical trade-offs. Higher resolutions improve cache specificity (higher chance that an incoming query is covered by already-hot cells and less overfetch when assembling answers) but increase:

- The number of keys per query.
- Metadata/memory overhead per cached entry.
- Composition cost when merging many cell payloads.

Lower resolutions reduce key churn and memory but risk overestimation (returning many features outside the exact footprint) and lower hit probability for small, detailed queries. The evaluation therefore considers a small set of candidate resolutions and reports hit ratio, composition overhead, and memory per resolution, based on documented cell-area curves.

### 3.2.5 Mapping BBoxes/Polygons to Cells

Request footprints are consistently converted into H3 cell sets. A rectangular BBOX is treated as a four-corner polygon, while user-defined polygons are used directly. The conversion uses H3’s region functions: given a polygon and a target resolution, return the set of cell IDs whose centroids lie inside the polygon [37]. This centroid-containment rule is explicit in both the H3 reference and in SQL integrations. The inverse operation reconstructs polygon outlines from a set of cells, which is used for debugging and visualization [43].

Boundary effects must be handled carefully. Because containment is based on cell centroids, narrow polygons may miss edge-touching cells whose centers fall just outside, while low-resolution cells can over-cover the footprint. These issues are handled by choosing a resolution appropriate to the query’s spatial detail, optionally expanding the boundary by a 1-ring and clipping results to the original geometry, and always reapplying the original filters when combining cached cells to avoid false positives. For multipolygons and holes, the outer ring and interior rings are passed to the region function so that coverage excludes holes and unconnected islands are handled correctly. Neighboring cells are included only when a buffer around the footprint is explicitly needed for later rendering or prefetching [35].

### 3.2.6 Spatial Approximation Limits of H3

Converting continuous geometries into hexagonal cells introduces approximation errors that must be considered during design and evaluation [24]. First, *centroid-based coverage* in common `polygonToCells`/`polyfill` routines only includes cells whose centres fall inside the polygon [37]; thin corridors or sliver-like shapes near boundaries can therefore be under-covered unless the resolution is increased or a slight expansion step is added. Second, the H3 grid is constructed on an icosahedral projection, which gives small but systematic *edge misalignments* and area variation across the globe, cells near certain parts of the icosahedron deviate more from equal-area assumptions [42], and reconstructed boundaries from cell sets may appear slightly jagged compared to the original geometry. Third, *pentagon distortion* creates uncommon neighbourhoods where

rings/ranges have special handling, algorithms that rely on uniform six-neighbour topology can fail or require slower fallback paths around pentagons [35].

Practical mitigations used in this thesis are:

- Choose a resolution appropriate to the query footprint.
- Optionally expand by a 1-ring at the target resolution and trim server-side.
- Always reapply the exact PostGIS spatial condition on composed results.
- Handle cells near pentagon distortion zones carefully during experiments.

Together, these measures bound the approximation error while preserving most of the performance benefit of cell-based reuse.

### 3.3 Caching Theory for Spatial Workloads

#### 3.3.1 Caching Objectives and Cost Model

The cache objectives are defined in terms of latency and backend offload. Let  $H$  be the cache hit ratio for a given configuration (resolution, admission, replacement). Let  $L_{\text{hit}}$  be the end-to-end latency on a hit (middleware lookup + composition + optional post-filter), and  $L_{\text{miss}}$  the latency on a miss (database + network + rendering). The expected latency is

$$\mathbb{E}[L] = H L_{\text{hit}} + (1 - H) L_{\text{miss}}.$$

Similarly, the backend work (e.g., CPU/IO in PostGIS) is modeled with per-request costs  $B_{\text{hit}}$  and  $B_{\text{miss}}$  (typically  $B_{\text{hit}} \ll B_{\text{miss}}$ ), giving

$$\mathbb{E}[B] = H B_{\text{hit}} + (1 - H) B_{\text{miss}}.$$

A simple overall objective can be expressed as a weighted sum

$$J = \alpha \mathbb{E}[L] + \beta \mathbb{E}[B] + \gamma S,$$

where  $S$  measures staleness risk (e.g., fraction of responses older than a freshness target) and  $(\alpha, \beta, \gamma)$  reflect deployment priorities. In practice, the tail latency (P95/P99) is also reported, because heavy-tailed request sizes and composition can make percentiles differ significantly from the mean [7]. Under this model, policies are compared by how they trade hit ratio and composition overhead against memory footprint, while respecting a freshness bound set by TTL and event-driven invalidation.

#### 3.3.2 Key Design for Spatial Queries

Cache keys define how granular and isolated cache reuse is. A composite key that captures all the information needed is used to guarantee matches with a database answer:

`key = <layer> : <srid> : <resolution> : <cellId> : <filterHash> : <version>.`

Here `layer` identifies the dataset; `srid` fixes the coordinate space; `resolution/cellId` identify the H3 bucket; `filterHash` fingerprints the attribute condition (normalised ECQL/FES); and `version` encodes schema/data epoch for bulk invalidation. Multi-cell requests are handled by fetching the per-cell entries and composing them, then reapplying the original predicate to avoid overfetching extra features.

This per-cell design avoids generating a large number of unique keys for complex polygons, improves reuse because nearby cells often appear in many queries, and allows targeted invalidation when upstream changes touch only a subset of cells. Larger “query keys” (entire cell-sets) can be layered on top as an optimisation for extremely frequent, identical windows, but the base unit of reuse remains the single cell.

### 3.3.3 Admission and Replacement

Since spatial workloads are uneven, the cache should prioritise cells that remain popular over time while avoiding frequent replacements. A hotness-based admission rule is used: each cell keeps a fading frequency counter, and new items are stored only if recent access passes a set limit or if they replace a clearly less active item already in the cache. Lightweight frequency estimators (for example, TinyLFU-style) give small, memory-efficient estimates that work well under Zipf-like access patterns.

For replacement, both how recent and how frequent accesses are considered to protect cells that are either short-term spikes or long-term hotspots. In practice, an ARC-style split between recent and frequent lists adapts to changing traffic patterns [20], while the admission rule prevents one-time scans from filling up the cache. Spatially aware strategies include:

- Group removal: when memory becomes limited, remove clusters of nearby, cold cells at the same resolution to avoid uneven gaps.
- Resolution aware fallback: before removing many small, high-detail cells, try combining them into a larger cell if it still provides good reuse.
- Ghost records (metadata only): that remember removed cells to guide future admission choices [20].

These methods keep the active cache focused on real hotspots and help the hit rate stay steady even when demand patterns change.

### 3.3.4 TTL-Based Freshness Control

A time-to-live (TTL) sets an upper bound on how long a cached cell may be reused without revalidation. TTL follows the same idea as the HTTP *freshness lifetime*: a response is *fresh* while its age is below the set limit and becomes *stale* afterward. In the middleware, each Redis entry stores its own TTL, and expiration is handled by Redis through active/idle checks and lazy deletion on access. Let  $\tau$  be the TTL and assume upstream edits arrive as a Poisson process with rate  $\lambda$  for a given cell. Then the probability that a cached answer is stale at access time is  $P_{\text{stale}} = 1 - e^{-\lambda\tau}$  [30], which makes the  $\lambda\tau$  product a practical tuning knob, where larger  $\tau$  increases hit ratio but also the risk of staleness.

Default TTLs are set per layer, and adaptive TTLs are supported: shorter  $\tau$  values are used for frequently updated layers (e.g., work orders) and longer ones for static layers (e.g., base maps). To cap worst-case staleness, clients can request *fresh* responses by bypassing the cache or forcing revalidation. Finally, TTL interacts with event-driven invalidation, so any update event that maps to a cell immediately removes or refreshes the entry, regardless of remaining TTL (see next subsection).

### 3.3.5 Event-Driven Invalidation

TTL alone cannot guarantee up-to-date data for regions that change often. As a result, the system subscribes to change-data-capture (CDC) streams from PostgreSQL via Debezium, which tails the



database WAL and sends an event per INSERT/UPDATE/DELETE to Kafka topics [8]. Each event includes table and row identifiers, and when geometries change, the middleware calculates which H3 cells are affected and invalidates their corresponding cache keys. This gives near-real-time consistency for updated features while avoiding global cache purges.

Because streaming systems provide ordering only within a partition, events are partitioned by stable identifiers (e.g., feature ID), so that all changes for the same feature arrive in order [3]; across features, out-of-order delivery is acceptable because invalidation is repeat-safe. Repeat-safe logic is implemented with a per-key *version/epoch* based on the source commit LSN; invalidating the same cell twice is safe, and late events with older LSNs are ignored [32]. Delivery guarantees are at-least-once by default, so when needed, Kafka’s repeat-safe producers/transactions enable exactly-once processing in consumers [6], though with higher operational cost. In practice, combining at-least-once delivery with repeat-safe, versioned invalidation is sufficient: duplicates only cause harmless extra removals, and missing entries are simply repopulated on the next access.

### 3.4 Consistency and Freshness Semantics

#### 3.4.1 Definitions: Consistency, Coherence, and Staleness

*Coherence* is a single-object property: all replicas of the same cached item eventually agree on the most recent value; in other words, two readers of the same key do not see conflicting versions once updates have spread [23]. *Consistency* refers to multi-object and temporal guarantees, e.g., whether reads reflect a globally ordered history of writes. Strong models such as linearizability make each operation appear atomic in real time [13], whereas weaker models (e.g., eventual consistency) allow temporary differences [45]. The middleware targets *bounded staleness* rather than full linearizability: bounded in time using TTL, and shortened with event-driven invalidation.

*Staleness* measures how “out of date” a cache response is relative to the origin. A response becomes *stale* once its age exceeds its freshness lifetime. Staleness is reported as a fraction of responses served beyond their freshness targets, and policies are preferred that keep this fraction below a layer-specific threshold while still delivering high hit ratios.

#### 3.4.2 Spatial Validity Scopes

In spatial workloads, every cached entry has to carry an explicit validity *scope*. Each value is tied to the H3 cell (or cell set) used as its key, and an update is relevant only if the updated geometry intersects that scope. This mirrors prior work on location-dependent caching, where invalidation combines temporal and spatial conditions to avoid discarding unrelated data. Formally, let  $V(k)$  be the validity polygon of key  $k$  (the cell footprint), and let  $\Delta$  be the geometry delta from an update event. Invalidate  $k$  when  $V(k) \cap \Delta \neq \emptyset$ .

Two edge cases matter in practice. First, boundary cells: to reduce false negatives due to discretization, the scope can be expanded by a one-ring buffer at the chosen resolution and the exact PostGIS filter can still be reapplied at compose time. Second, multi-cell requests: scopes compose as unions; invalidation touches only the intersecting cells, keeping cache content for unaffected neighbors. Together with TTL, these spatial scopes limit cache clearing and keep popular regions updated without emptying the entire cache.

#### 3.4.3 Combining TTL with Event-Driven Updates

A hybrid freshness strategy combines a *time-to-live* (TTL) with *event-driven* invalidation so that cached entries are reused aggressively yet replaced quickly when underlying data change. TTL provides a deterministic upper bound on age: a value is *fresh* while its age  $< \tau$  and *stale* after

that [14]. Event-driven invalidation reacts to specific updates (e.g., CDC messages) by removing or refreshing only the keys whose spatial scopes intersect the changed features. Let updates for a particular cell arrive according to a rate  $\lambda$ , and let  $D$  be the end-to-end delay from an upstream commit to the consumer performing invalidation. The worst-case staleness window is  $\min(\tau, D)$ , and the approximate probability that a randomly timed read will encounter stale data is  $1 - e^{-\lambda \min(\tau, \mathbb{E}[D])}$  [30]. Tuning is therefore done along two axes:

- Choosing  $\tau$  by layer volatility and tolerance for stale reads.
- Reducing  $\mathbb{E}[D]$  by provisioning streaming paths and consumers.

In practice, *refresh-on-read* for near-expiry hits is also included, a *write-through* path for updates made through the service itself, and short *grace* windows that allow slightly stale entries to be served while a background refresh runs, keeping tail latency low while still controlling staleness.

### 3.4.4 Delivery Semantics and Ordering

Event-driven invalidation relies on the messaging substrate’s guarantees. Kafka keeps order *within* a topic partition, so partitioning by a stable key (e.g., feature identifier or H3 cell) gives in-order change events for items mapped to the same key [3]. Default delivery is at least once, so consumers must therefore be repeat-safe. An ever-increasing source version (e.g., PostgreSQL LSN from Debezium) is attached to each event and *compare-and-set* invalidation is applied: drop cache entries if and only if the incoming version is newer than the last processed one. Duplicate or out-of-order events are therefore harmless [8]. Where required, Kafka’s reliable producers and transactions enable exactly-once processing for read-process-write topologies, trading operational complexity for stronger guarantees. Finally, recovery time after outages is limited by replaying from a saved offset and reapplying repeat-safe invalidations. Because invalidation is repeat-safe, replay restores consistency without global cache purges. Together, partitioned ordering, repeat-safe handlers, and version checks ensure a consistent, minimal-invalidations stream that keeps the cache aligned.

## 3.5 Performance and Latency Modeling

### 3.5.1 End-to-End Latency Decomposition

End-to-end latency is broken down into separate components to identify bottlenecks and target optimizations. Let a request traverse:

- The client  $\rightarrow$  middleware network.
- Middleware parsing/routing (Tmw).
- Cache lookup and (if hit) result composition plus optional post-filter.
- Database path on a miss—planner, index traversal, predicate evaluation, result materialization.
- Middleware  $\rightarrow$  client transfer plus serialization.

Then the total end-to-end latency  $T_{e2e}$  is

$$T_{e2e} = T_{\text{net,in}} + T_{\text{mw}} + \begin{cases} T_{\text{cache}} & \text{hit,} \\ T_{\text{db}} + T_{\text{cache(post)}} & \text{miss,} \end{cases} + T_{\text{net,out}}.$$

On hits,  $T_{\text{db}} = 0$  and  $T_{\text{cache}}$  dominates, while on misses,  $T_{\text{db}}$  typically dominates. Each segment is measured and percentiles (P50/P95/P99) are tracked rather than means, since heavy-tailed effects can hide user-visible slowdowns in averages [7]. Additional SRE "golden signals" (latency, traffic, errors, saturation) help correlate spikes [12] in  $T_{\text{db}}$  with database saturation, or rises in  $T_{\text{cache}}$  with composition overhead at very high H3 resolutions. This decomposition helps design experiments (e.g., resolution sweeps) and informs operational decisions such as whether to scale the middleware, database, or network.

### 3.5.2 Queueing Basics and Little's Law

Queueing theory connects arrival rate, concurrency, and delay. Little's Law states that the average number of in-flight requests  $N$  equals arrival rate  $\lambda$  times average response time  $R$ :  $N = \lambda R$  [1]. Therefore, at a fixed  $\lambda$ , reducing  $R$  (via cache hits) lowers concurrency and pressure on backend systems. For a single-server station with exponential arrival gaps and service times (M/M/1), service rate  $\mu = 1/S$  (mean service time  $S$ ) and utilization  $\rho = \lambda/\mu$ . Stability requires  $\rho < 1$ , and the expected response time is

$$R = \frac{S}{1 - \rho} = \frac{1/\mu}{1 - \lambda/\mu}$$

cf. [22]; this shows how delay grows nonlinearly as utilization approaches 1. In this context, the 'server' can represent the PostGIS node on cache misses or the middleware on cache hits; the model can be extended to multiple cores or nodes (e.g., M/M/ $k$ ). Arrival rate  $\lambda$  is estimated from observed traffic,  $R$  is measured per path (hit/miss), and concurrency limits and thread pools are sized so that  $\rho$  stays well below overload levels at target latencies. These formulas provide a simple way to plan capacity and highlight the trade-off between raising the hit ratio and improving database speed.

### 3.5.3 Tail Latency and Percentiles

Average latency can hide slow cases that have the biggest impact on how responsive a system feels to users. A small number of slow operations (for example, long network paths, cache misses, garbage collection pauses, or slow database queries) can build up across microservices and cause much longer end-to-end delays. For this reason, high-percentile service-level indicators (SLIs), mainly P95 and P99, are used as the main measures for interactive workloads. P95 shows how most users experience the system during load spikes, while P99 captures the slowest visible responses under normal conditions and is affected by extra delay added by combining multiple backend calls in one request. These percentiles help guide both design (for example, keeping cache merge time low and limiting how many backends a request touches) and operations (for example, alerting on slowdowns that do not change the average). P50, P95, and P99 are reported for the full path and for key parts (cache, middleware, database), and tail-focused goals (for example, "P99 < 200 ms") are used to test whether adaptive spatial caching improves responsiveness beyond average latency.

### 3.5.4 Hit Ratio vs. Latency/Throughput Trade-off

Caching reduces both latency and backend load through two effects: serving hits quickly from memory and not sending those requests to the database. Let  $H$  be the hit ratio. With average

hit and miss latencies  $L_{\text{hit}}$  and  $L_{\text{miss}}$ , the expected end-to-end latency is

$$\mathbb{E}[L] = H L_{\text{hit}} + (1 - H) L_{\text{miss}}.$$

Similarly, the expected backend work (CPU/I/O) is

$$\mathbb{E}[B] = H B_{\text{hit}} + (1 - H) B_{\text{miss}},$$

with  $B_{\text{hit}} \ll B_{\text{miss}}$ . Increasing  $H$  therefore lowers  $\mathbb{E}[L]$  and directly offloads the database. Under Little’s Law, reducing  $R$  (response time) also lowers in-flight concurrency at the backend, improving queueing and tail latency [34]. However, raising  $H$  by pushing to very small spatial cells can add extra merge work and frequent key changes, which raises  $L_{\text{hit}}$  and memory pressure. The optimal point balances:

- Admission/replacement that prioritise reusable cells.
- Choosing a cell size that keeps reuse high while limiting how many cells each request needs.
- Freshness policies (TTL/events) that avoid invalidating hot cells too aggressively.

The evaluation therefore reports  $(H, \mathbb{E}[L], \text{P95/P99}, \mathbb{E}[B])$  together to show these trade-offs.

### 3.5.5 Granularity vs. Memory Cost (H3 Resolution)

H3 offers a hierarchy of resolutions: as the resolution increases, cells become smaller, and the number of cells needed to cover the same area grows roughly exponentially. Finer resolutions give better spatial accuracy—queries overlap more with cached cells, and extra data fetched outside the target area decreases—raising the chance of cache hits for small, detailed regions. The trade-off is more keys per request, higher metadata cost per cell, and longer merge times when combining many cell results. Coarser resolutions show the opposite effect: fewer keys, smaller index structures, and faster merging, but more overcoverage near boundaries and a lower chance that small queries match popular cells. Memory use also grows with the number of stored (layer, resolution, cell, filter) entries; at high resolutions, busy urban areas can spread across many nearby cells unless caching is selective. In practice, a few resolutions are tested and the one that gives the best overall results is chosen, with the highest P95/P99 improvement under acceptable memory use and merge cost. When hotspots cover multiple cells, a coarser “umbrella” entry can also be stored for bursts of traffic, while a subset of fine cells is kept for high-detail queries.

### 3.5.6 Workload Skew and Hotspot Dynamics

Spatial access patterns are usually uneven; a small number of regions get most of the traffic while the rest stay cold. Also, popularity changes over time with events or daily activity cycles. For this reason, per-cell activity is tracked using *time-aware* counters instead of raw counts. A sliding window over recent requests captures short-term demand and reacts quickly to changes, but it may forget too quickly. In contrast, *exponentially decayed* counts adapt more smoothly by giving less weight to older accesses using a decay factor  $\alpha \in (0, 1)$ . Cells are then admitted when their decayed frequency passes a threshold, compared to their memory cost and merge time. Replacement keeps long-term hotspots (high long-term weight) while letting short-lived spikes use space briefly in the “recent” list. To reduce uneven coverage, mild *spatial smoothing* is applied, where nearby cells inherit a small share of a cell’s activity, helping to catch slightly shifted viewports. Finally, to handle sudden bursts (for example, a city-wide event), the number of new cells that can be added in one interval is limited and coarser cells are temporarily promoted, merging many small keys into one while keeping fine-grained entries that stay consistently active.

### 3.5.7 Golden Signals and Cache Metrics

Standard SRE "golden signals" are adopted together with cache-specific metrics for evaluation and operations:

- **Latency:** P50/P95/P99 for end-to-end, cache-hit path, and miss path.
- **Throughput (Traffic):** requests/s overall, hits/s, misses/s; per-layer breakdown.
- **Errors:** non-2xx/5xx rates and cache/middleware exceptions.
- **Saturation:** CPU, memory, and connection pool utilisation for middleware and PostGIS; Redis memory/evictions.
- **Cache ratios:** hit/miss ratio, byte hit ratio, and origin offload (misses avoided).
- **Freshness:** fraction of responses older than target; invalidations/s; average delay from update to invalidate.
- **Spatial stats:** per-cell request share (skew), active-key count, keys per request, and average bytes per cell.

These metrics connect design choices (resolution, admission/replacement, TTL/events) to user outcomes (tail latency) and system health (backend offload and saturation).

## 4 Method

### 4.1 Research Approach and Study Design

The goal is to measure how an H3-keyed middleware cache affects latency and load under skewed spatial workloads, compared to no cache. Each configuration is run with the same datasets, the same mix of query footprints (BBOX and polygons), and the same arrival pattern. Experiments are containerised and scripted so runs are repeatable.

For each workload profile the configurations are cycled through, runs are repeated, and median and percentile metrics are reported across repetitions. External factors are fixed as much as possible (hardware, network, versions, etc.), warm up the system, and then collect measurements during a steady window. The SRE “golden signals” are reported and cache-specific indicators: P50/P95/P99 latency (overall, hit-path, miss-path), requests/s, error rate, saturation (CPU/memory), and hit/miss/partial-hit ratios.

Independent variables are H3 resolution (7–9), decay, and freshness policy. Dependent variables are latency percentiles, PostGIS CPU, Redis memory and evictions, and staleness.

Validity measures include repeated trials, fixed seeds for traffic, and cross-checking results. Threats to validity are noted (dataset representativeness, synthetic vs. real traffic, and Docker networking differences), and the evaluation is framed as comparative rather than absolute.

### 4.2 System Under Study and Assumptions

The system mirrors a common geospatial setup: GeoServer exposes WFS/WMS endpoints backed by PostGIS, a middleware proxy sits in front and integrates Redis for caching and Kafka for change events. Prometheus scrapes metrics from services. Clients issue WFS `GetFeature` requests with BBOX or polygon filters and optional attribute filters.

Layers are published in EPSG:4326 and responses are encoded as GeoJSON. Queries target urban layers with realistic skew, mixing small viewports (city blocks) with medium windows (districts). The cache key encodes (layer, SRID, resolution, H3 cell ID, filter hash, version). Each value has a TTL and can be invalidated by events. Redis is configured with a bounded memory and an eviction policy [33].

Kafka provides in-partition ordering [4]; consumers are in a single group per environment so each partition is processed by exactly one consumer. Event handling is repeat-safe so duplicate invalidations are harmless.

A stable network is assumed during testing, and no background load is assumed except the experiment. GeoServer/PostGIS schemas or indexes are not changed; the middleware is non-invasive and standards-compliant on the wire.

### 4.3 Middleware Design and Implementation

The middleware is a Go service that acts as an HTTP reverse proxy with spatial awareness. It parses incoming requests, maps footprints to H3 cells, looks up Redis for per-cell entries, forwards misses to GeoServer, and composes responses. A Kafka consumer listens for change events, maps updated geometries to cells, and deletes affected keys. The service exposes Prometheus metrics and health checks. Configuration is via environment variables and a simple YAML file; all components run in Docker for reproducibility.

#### 4.3.1 End-to-End Request Flow and Component Responsibilities

**1. Router:** Accepts client WFS/WMS requests, validates required parameters (service, request, typeNames, spatial filter), normalises ECQL/CQL filters, and builds an internal QueryRequest. It preserves wire compatibility with GeoServer so clients do not change their integrations.

**2. H3 mapper:** Converts the footprint (BBOX as polygon, or user polygon) to a set of H3 cell IDs at the chosen resolution. The selection of resolution is static per run.

**3. Decision and hotness:** Updates per-cell counters (exponential decay). Cells above threshold are eligible for caching; others pass through without admission to reduce turnover. This keeps memory focused on regions with reuse.

**4. Cache manager:** Builds composite keys (layer/SRID/res/cell/filter-hash/version), performs a single MGET to Redis, and classifies the request as full hit, partial hit, or miss. Values are stored with TTL. Redis is configured with an eviction policy suitable for caches and a fixed maxmemory to force realistic trade-offs.

**5. Executor:** For cells not found, constructs a WFS `GetFeature` URL and forwards to GeoServer. The result is parsed, stored per-cell (with TTL), and streamed to the aggregator.

**6. Aggregator:** Merges per-cell payloads, deduplicates features by stable identifiers, reapplies the original spatial/attribute filter to avoid overfetch, and returns a single GeoJSON response. It records keys-per-request and merge time.

**7. Invalidation listener:** Subscribes to Kafka topics carrying change events. For each event, maps the changed geometry to H3 cells and issues key deletions. Processing is idempotent, and ordering is ensured within partitions.

**8. Metrics/health:** Exposes request counters, latency histograms, hit/miss ratios, Redis stats, and invalidation lag. Health endpoints return liveness/readiness for containers.

#### 4.3.2 H3 Footprint Mapping and Request Bucketing

H3's polygon-to-cells ("polygonToCells") functions are used to map a request footprint to cell IDs [37]. A BBOX is treated as a polygon with four vertices. For each request, the mapper returns the set of cell IDs at resolution  $r$ , which become the unit of reuse and invalidation. The default polygonToCells uses a centroid-in-polygon rule [37], which is simple and fast; thin shapes near boundaries can be handled by increasing  $r$  or by expanding one ring and trimming on compose.

Resolution controls granularity. Higher  $r$  gives smaller cells, more precise coverage, and often higher hit probability for small windows, but increases keys per request. Lower  $r$  reduces keys and metadata but can over-cover and reduce locality. Resolution is therefore swept with  $r \in \{5, 6, 7, 8, 9, 10\}$  in experiments and hit ratio, keys-per-request, and bytes-per-cell are recorded.

The cache key includes the cell ID and a stable hash of the attribute filter (normalised ECQL). This keeps semantically different queries isolated while enabling reuse when both footprint and filter match.

On composed responses the original spatial filter is reapplied to remove any features picked up by boundary cells. Multi-polygons are supported by passing outer and inner rings to the polyfill. All cell operations are deterministic so repeated queries map to the same key set.

#### 4.3.3 Caching in Redis: Key Schema and Result Storage

The cache entries are stored in Redis using a composite key: `<layer>:<srId>:<res>:<cellId>:<filterHash>:<version>`. Values are kept as Redis strings (binary-safe blobs), which allows compact, compressed payloads without concerns about encoding. Each request that spans multiple H3 cells triggers a single MGET for those keys to reduce round trips. `cellId` is the H3 64-bit

identifier [40], and `filterHash` fingerprints the normalised ECQL/FES filter so semantically different queries do not collide.

The value layout is simple: a small header (timestamps, byte length, schema/version) and a payload. For vector responses GeoJSON Features or small, per-cell arrays of feature rows are stored. TTL is set at write and attached per key, Redis removes expired keys automatically. Memory pressure is controlled by `maxmemory` and an eviction policy (e.g., `allkeys-lru` or `allkeys-lfu`) so the cache behaves predictably when full.

#### 4.3.4 Response Composition and Correctness Safeguards

The aggregator merges per-cell payloads into one `FeatureCollection`. De-duplication is performed by a stable feature identifier (primary key) that is also exposed as the GeoJSON `id` when possible; this avoids duplicates when neighbouring cells overlap. After merging, the original attribute and spatial filter are reapplied so the final result matches what PostGIS would have returned for the exact footprint. In practice exact geometry checks (e.g., `ST_Intersects`/`ST_Within`) are applied on the composed set to trim any over-coverage from boundary cells.

Partial failures are also guarded against. If some cells miss or expire while others hit, the middleware fetches the missing ones from GeoServer and only returns once the full set is assembled. Latencies and correctness counters are tagged by “full-hit/partial/miss” so the impact in P50/P95/P99 can be inspected and post-filtering can be verified to keep answers faithful to the original query.

### 4.4 Freshness Mechanisms

TTL-based freshness is combined with event-driven invalidation. TTL gives a clear freshness lifetime (fresh vs. stale) that is easy to reason about and measure. Event-driven invalidation reacts to actual data changes and targets only the overlapping H3 cells, keeping hot regions up to date without broad purges.

#### 4.4.1 TTL Policy and Configuration

On write, each key gets a TTL, keys expire automatically once the countdown reaches zero. TTL is set at insert time. Remaining lifetime is tracked with `TTL/PTTL` during debugging. TTLs are per layer: fast-changing layers get short TTLs, static base layers get longer ones.

Operationally, Redis removes expired keys and can also evict keys under memory pressure according to `maxmemory-policy`. An *allkeys* policy (LRU/LFU) is used when the dataset is entirely cacheable and memory-bounded, or a *volatile* policy is used when only TTL-tagged entries should be evicted. Metrics for expiry and eviction are also exposed so freshness decisions (TTL) can be separated from memory backpressure effects (evictions) in the results.

#### 4.4.2 Kafka/CDC Event Processing and Spatial Invalidation

To cut staleness after updates, change events are consumed from PostgreSQL. The PostgreSQL connector streams inserts/updates/deletes from the WAL as structured messages; the middleware extracts the changed geometry, maps it to H3 cells, and invalidates the matching keys. This targets only the affected spatial scope and avoids global cache clears.

Kafka’s ordering within a partition is relied on: by partitioning events on a stable key (e.g., feature ID), all changes for the same feature arrive in order. Delivery is at-least-once by default [2], so the invalidation handler is *idempotent* and *version-aware* (drops events that have already been processed at an equal or newer source LSN). Where a stricter pipeline is needed, Kafka’s



idempotent producers and transactions can be used to enable exactly-once processing, with added operational cost. These guarantees keep invalidation consistent even during retries or restarts.

For spatial targeting the H3 cells touched by the changed geometry are computed using the standard polygon-to-cells functions, then keys for those `<layer,res,cellId,filterHash>` combinations are deleted. This ties freshness directly to spatial overlap while keeping runtime costs low.

## 4.5 Experimental Methodology

The experiments compare an H3-keyed middleware cache against baselines under controlled, repeatable conditions. Each run follows a fixed script: container start, warm-up, timed measurement window, and teardown. Image tags and record digests are pinned to keep versions stable across repetitions. Load is driven by scripted scenarios at a fixed arrival rate so that request mix and intensity are identical between configurations. Runs are repeated multiple times, and then aggregate medians and percentiles across repetitions.

Measurements are taken only during a steady window after warm-up. Latency (P50/P95/P99), hit/miss rates, and backend load (PostGIS CPU, Redis memory/evictions) are collected. External factors are kept stable: hardware, container limits, network, and dataset. The study is comparative rather than absolute, so the study does not claim universal latency numbers, but it does quantify deltas between configurations under the same workload. Threats to validity (synthetic traffic, single-node deployments, and Docker networking) are noted, and they are mitigated by repeating trials with fixed seeds.

### 4.5.1 Testbed and Deployment (Containers, Versions, Hardware)

All components run in Docker on a single host laptop. Images are pinned by tag and the exact image digest is recorded in the repository. The stack includes: GeoServer (WFS/WMS), PostgreSQL, PostGIS, Redis, the middleware proxy, Kafka, Prometheus, Grafana, Loki, and Promtail. Each service is launched via a compose file. The dataset is loaded once at startup, and containers are restarted between configurations to reset caches.

Hardware was a single laptop running Arch Linux (Lenovo Legion 7 15IMH05) with an Intel Core i7-10750H (6 cores / 12 threads) and 16 GiB RAM. The experimental stack ran in Docker containers (PostGIS, GeoServer, Redis, Kafka, and monitoring). Redis was configured with a fixed memory budget using `maxmemory=256 MiB` and the `allkeys-lru` eviction policy to bound cache growth. Prometheus scraped service metrics and Grafana dashboards summarized latency distributions, throughput, error rates, and Redis memory/evictions; log-based observability was provided via Loki/Promtail.

### 4.5.2 Baselines and Compared Configurations

Four configurations are compared:

1. **Baseline with low Zipf:** No cache, requests go directly to GeoServer/PostGIS, and the distribution of calls are spread out.
2. **Baseline with high Zipf:** No cache, requests go directly to GeoServer/PostGIS, and the distribution of calls are compact.
3. **H3 cache with low H3 res and low Zipf:** per-cell keys at low H3 resolution, and the distribution of calls are spread out.

4. **H3 cache with high H3 res and low Zipf**: per-cell keys at high H3 resolution, and the distribution of calls are spread out.
5. **H3 cache with low H3 res and high Zipf**: per-cell keys at low H3 resolution, and the distribution of calls are compact.
6. **H3 cache with high H3 res and high Zipf**: per-cell keys at high H3 resolution, and the distribution of calls are compact.

For H3 resolutions  $r \in \{5, 6, 7, 8, 9, 10\}$  are used to study hit ratio, memory, speed, and correctness. For each configuration the same dataset, arrival pattern, and container limits are kept. End-to-end latency (P50/P95/P99), misses avoided, Redis memory/evictions, and staleness (fraction of responses older than target) are reported.

#### 4.5.3 Experiment Procedure

Each trial proceeds as follows:

1. Start all containers and wait for health checks to pass.
2. Warm up the system by running the target request for a fixed duration (e.g., 1 minutes) to stabilise caches and JIT optimisations.
3. Run the load generator at the target constant arrival rate for a fixed measurement window (e.g., 4 minutes), collecting metrics during this period.
4. Stop the load generator and export collected metrics from Prometheus and middleware logs for analysis.
5. Restart containers to reset state before the next configuration.

Repeat this procedure for each configuration and workload, ensuring that each run uses the same request list and arrival pattern. Each configuration is repeated multiple times (e.g., 5 times) to account for variability, and median/percentile metrics are computed across runs.

## 5 Results

x

## 6 Discussion

x

## 7 Conclusions

x

## References

- [1] “A Proof for the Queuing Formula:  $L = \lambda W$ ”. In: ().
- [2] Apache Software Foundation. *Apache Kafka 4.1: Design*. <https://kafka.apache.org/41/design/design/>. Apache Kafka Project, 2025. URL: <https://kafka.apache.org/41/design/design/>.
- [3] Apache Software Foundation. *Apache Kafka Documentation*. <https://kafka.apache.org/documentation/>. Apache Kafka Project, 2025. URL: <https://kafka.apache.org/documentation/>.
- [4] Apache Software Foundation. *Introduction to Apache Kafka*. <https://kafka.apache.org/24/getting-started/introduction/>. Apache Kafka Project, 2024. URL: <https://kafka.apache.org/24/getting-started/introduction/>.
- [5] ClickHouse, Inc. *H3 Geospatial Functions: ClickHouse SQL Reference*. <https://clickhouse.com/docs/sql-reference/functions/geo/h3>. ClickHouse, Inc., 2025. URL: <https://clickhouse.com/docs/sql-reference/functions/geo/h3>.
- [6] Confluent, Inc. *Kafka Design: Delivery Semantics*. <https://docs.confluent.io/kafka/design/delivery-semantics.html>. Confluent Documentation, 2025. URL: <https://docs.confluent.io/kafka/design/delivery-semantics.html>.
- [7] Jeffrey Dean and Luiz André Barroso. “The tail at scale”. In: *Communications of the ACM* 56.2 (Feb. 2013), pp. 74–80. ISSN: 1557-7317. DOI: 10.1145/2408776.2408794. URL: <http://dx.doi.org/10.1145/2408776.2408794>.
- [8] Debezium Community. *Debezium PostgreSQL Connector Documentation*. <https://debezium.io/documentation/reference/stable/connectors/postgresql.html>. Debezium Project, 2025. URL: <https://debezium.io/documentation/reference/stable/connectors/postgresql.html>.
- [9] Nicolas Drapier, Aladine Chetouani, and Aurélien Chateigner. “Enhancing Maritime Trajectory Forecasting via H3 Index and Causal Language Modelling (CLM)”. In: *Transactions on Machine Learning Research* (2024). DOI: 10.48550/ARXIV.2405.09596. URL: <https://arxiv.org/abs/2405.09596>.
- [10] GeoSolutions. *Tile Caching with GeoWebCache*. GeoServer Training (official documentation). 2020. URL: <https://docs.geoserver.geo-solutions.it/edu/en/enterprise/gwc.html>.
- [11] GeoVista Developers. *Spatial Index Example: Uber H3 Integration*. [https://geovista.readthedocs.io/en/stable/generated/gallery/spatial\\_index/uber\\_h3.html](https://geovista.readthedocs.io/en/stable/generated/gallery/spatial_index/uber_h3.html). GeoVista Project, 2025. URL: [https://geovista.readthedocs.io/en/stable/generated/gallery/spatial\\_index/uber\\_h3.html](https://geovista.readthedocs.io/en/stable/generated/gallery/spatial_index/uber_h3.html).
- [12] Google SRE Team. “Monitoring Distributed Systems”. In: *Site Reliability Engineering: How Google Runs Production Systems*. O’Reilly Media, 2016. URL: <https://sre.google/sre-book/monitoring-distributed-systems/>.
- [13] Maurice P. Herlihy and Jeannette M. Wing. “Linearizability: a correctness condition for concurrent objects”. In: *ACM Transactions on Programming Languages and Systems* 12.3 (July 1990), pp. 463–492. ISSN: 1558-4593. DOI: 10.1145/78969.78972. URL: <http://dx.doi.org/10.1145/78969.78972>.
- [14] *HTTP Caching*. June 2022. DOI: 10.17487/rfc9111. URL: <http://dx.doi.org/10.17487/RFC9111>.

- [15] Andreas Kipf et al. *Adaptive Main-Memory Indexing for High-Performance Point-Polygon Joins*. en. 2020. DOI: 10.5441/002/EDBT.2020.31. URL: [https://openproceedings.org/2020/conf/edbt/paper\\_190.pdf](https://openproceedings.org/2020/conf/edbt/paper_190.pdf).
- [16] Lauro Lins, James T. Klosowski, and Carlos Scheidegger. “Nanocubes for Real-Time Exploration of Spatiotemporal Datasets”. In: *IEEE Transactions on Visualization and Computer Graphics* 19.12 (Dec. 2013), pp. 2456–2465. ISSN: 1077-2626. DOI: 10.1109/tvcg.2013.179. URL: <http://dx.doi.org/10.1109/TVCG.2013.179>.
- [17] Chang Liu, Brendan C. Fruin, and Hanan Samet. “SAC: semantic adaptive caching for spatial mobile applications”. In: *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. SIGSPATIAL’13. ACM, Nov. 2013, pp. 174–183. DOI: 10.1145/2525314.2525366. URL: <http://dx.doi.org/10.1145/2525314.2525366>.
- [18] Helo  se Manica, M. A. R. Dantas, and Murilo S. de Camargo. “An Architecture for Location-Dependent Semantic Cache Management”. In: *Proceedings of the Seventh International Conference on Enterprise Information Systems*. SciTePress: Science, 2005, pp. 320–325. DOI: 10.5220/0002524903200325. URL: <http://dx.doi.org/10.5220/0002524903200325>.
- [19] Stavros Maroulis et al. “Visualization-Aware Time Series Min-Max Caching with Error Bound Guarantees”. In: *Proceedings of the VLDB Endowment* 17.8 (Apr. 2024), pp. 2091–2103. ISSN: 2150-8097. DOI: 10.14778/3659437.3659460. URL: <http://dx.doi.org/10.14778/3659437.3659460>.
- [20] Nimrod Megiddo and Dharmendra S. Modha. “ARC: A Self-Tuning, Low Overhead Replacement Cache”. In: *2nd USENIX Conference on File and Storage Technologies (FAST 03)*. San Francisco, CA: USENIX Association, Mar. 2003. URL: <https://www.usenix.org/conference/fast-03/arc-self-tuning-low-overhead-replacement-cache>.
- [21] Saptashwa Mitra et al. “STASH: Fast Hierarchical Aggregation Queries for Effective Visual Spatiotemporal Explorations”. In: *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, Sept. 2019, pp. 1–11. DOI: 10.1109/cluster.2019.8891029. URL: <http://dx.doi.org/10.1109/CLUSTER.2019.8891029>.
- [22] Daniel S. Myers. *Lecture 12: The M/M/1 Queue*. [https://pages.cs.wisc.edu/~dsmyers/cs547/lecture\\_12\\_mm1\\_queue.pdf](https://pages.cs.wisc.edu/~dsmyers/cs547/lecture_12_mm1_queue.pdf). University of Wisconsin–Madison, CS 547 Lecture Notes [Online; accessed 22-October-2025]. 2017.
- [23] Vijay Nagarajan et al. *A Primer on Memory Consistency and Cache Coherence*. Springer International Publishing, 2020. ISBN: 9783031017643. DOI: 10.1007/978-3-031-01764-3. URL: <http://dx.doi.org/10.1007/978-3-031-01764-3>.
- [24] Open Geospatial Consortium. *OGC API: Features, Part 1: Core*. Tech. rep. OGC 20-040r3. Open Geospatial Consortium, 2023. URL: <https://docs.ogc.org/as/20-040r3/20-040r3.pdf>.
- [25] Open Geospatial Consortium. *OGC Filter Encoding 2.0 Encoding Standard*. Tech. rep. OGC 09-026r2. Open Geospatial Consortium, 2011.
- [26] Open Geospatial Consortium. *Web Feature Service (WFS) Standard*. <https://www.ogc.org/standards/wfs/>. 2025.
- [27] Open Geospatial Consortium. *Web Map Tile Service (WMTS) Standard*. <https://www.ogc.org/standards/wmts/>. 2025.

- [28] Shaoming Pan et al. “A Global User-Driven Model for Tile Prefetching in Web Geographical Information Systems”. In: *PLOS ONE* 12.1 (Jan. 2017). Ed. by Houbing Song, e0170195. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0170195. URL: <http://dx.doi.org/10.1371/journal.pone.0170195>.
- [29] Dimitris Papadias et al. “Efficient OLAP Operations in Spatial Data Warehouses”. In: *Advances in Spatial and Temporal Databases*. Springer Berlin Heidelberg, 2001, pp. 443–459. ISBN: 9783540477242. DOI: 10.1007/3-540-47724-1\_23. URL: [http://dx.doi.org/10.1007/3-540-47724-1\\_23](http://dx.doi.org/10.1007/3-540-47724-1_23).
- [30] Hossein Pishro-Nik. *Basic Concepts of the Poisson Process*. 2014. URL: [https://www.probabilitycourse.com/chapter11/11\\_1\\_2\\_basic\\_concepts\\_of\\_the\\_poisson\\_process.php](https://www.probabilitycourse.com/chapter11/11_1_2_basic_concepts_of_the_poisson_process.php).
- [31] Paul Ramsey, Mark Leslie, and PostGIS contributors. *Spatial Indexing: Introduction to PostGIS*. <https://postgis.net/workshops/postgis-intro/indexing.html>. 2012–2023.
- [32] Red Hat, Inc. *Debezium Connector for PostgreSQL: Red Hat Build of Debezium 2.1.4 User Guide*. [https://docs.redhat.com/en/documentation/red\\_hat\\_build\\_of\\_debezium/2.1.4/html/debezium\\_user\\_guide/debezium-connector-for-postgresql](https://docs.redhat.com/en/documentation/red_hat_build_of_debezium/2.1.4/html/debezium_user_guide/debezium-connector-for-postgresql). Red Hat Documentation, 2025. URL: [https://docs.redhat.com/en/documentation/red\\_hat\\_build\\_of\\_debezium/2.1.4/html/debezium\\_user\\_guide/debezium-connector-for-postgresql](https://docs.redhat.com/en/documentation/red_hat_build_of_debezium/2.1.4/html/debezium_user_guide/debezium-connector-for-postgresql).
- [33] year = 2025 howpublished = \bibnamedelimb<https://redis.io/docs/latest/develop/reference/eviction/> organization = Redis url = <https://redis.io/docs/latest/develop/reference/eviction/> Redis title = Key Eviction — Redis Reference.
- [34] Gennady Samorodnitsky. *Little’s Law: Stochastic Processes I Lecture Notes*. <https://www.columbia.edu/~ks20/stochastic-I/stochastic-I-LL.pdf>. 2018.
- [35] Uber Technologies, Inc. *H3 API: Grid Traversal Functions*. <https://h3geo.org/docs/api/traversal/>. H3 Project, 2025. URL: <https://h3geo.org/docs/api/traversal/>.
- [36] Uber Technologies, Inc. *H3 API: Hierarchy Functions*. <https://h3geo.org/docs/api/hierarchy/>. H3 Project, 2025. URL: <https://h3geo.org/docs/api/hierarchy/>.
- [37] Uber Technologies, Inc. *H3 API: Region Functions*. <https://h3geo.org/docs/api/regions/>. H3 Project, 2025. URL: <https://h3geo.org/docs/api/regions/>.
- [38] Uber Technologies, Inc. *H3 API: Unidirectional Edge Functions*. <https://h3geo.org/docs/api/uniedge/>. H3 Project, 2025. URL: <https://h3geo.org/docs/api/uniedge/>.
- [39] Uber Technologies, Inc. *H3 Core Library: Average Cell Area by Resolution*. <https://h3geo.org/docs/core-library/restable/#average-area-in-km2>. H3 Project, 2025. URL: <https://h3geo.org/docs/core-library/restable/%5C#average-area-in-km2>.
- [40] Uber Technologies, Inc. *H3 Core Library: H3 Indexing*. <https://h3geo.org/docs/core-library/h3Indexing/>. H3 Project, 2025. URL: <https://h3geo.org/docs/core-library/h3Indexing/>.
- [41] Uber Technologies, Inc. *H3 Core Library: Overview*. <https://h3geo.org/docs/core-library/overview/>. H3 Project, 2025. URL: <https://h3geo.org/docs/core-library/overview/>.
- [42] Uber Technologies, Inc. *H3 Core Library: Resolution Table*. <https://h3geo.org/docs/core-library/restable/>. H3 Project, 2025. URL: <https://h3geo.org/docs/core-library/restable/>.



- [43] Uber Technologies, Inc. *H3-py Tutorial: Working with Polygons*. [https://uber.github.io/h3-py/polygon\\_tutorial.html](https://uber.github.io/h3-py/polygon_tutorial.html). Uber Technologies, Inc., 2025. URL: [https://uber.github.io/h3-py/polygon\\_tutorial.html](https://uber.github.io/h3-py/polygon_tutorial.html).
- [44] Uber Technologies, Inc. *H3: A Hexagonal Hierarchical Spatial Index*. <https://h3geo.org/docs/>. 2025.
- [45] Werner Vogels. “Eventually consistent”. In: *Communications of the ACM* 52.1 (Jan. 2009), pp. 40–44. ISSN: 1557-7317. DOI: 10.1145/1435417.1435432. URL: <http://dx.doi.org/10.1145/1435417.1435432>.
- [46] Yiwen Wang. “Vecstra: An Efficient and Scalable Geo-spatial In-Memory Cache”. In: *Proceedings of the VLDB 2018 Ph.D. Workshop, August 27, 2018, Rio de Janeiro, Brazil*. CEUR Workshop Proceedings, 2018. URL: <https://ceur-ws.org/Vol-2175/paper06.pdf>.
- [47] Christian Winter et al. “GeoBlocks: A Query-Cache Accelerated Data Structure for Spatial Aggregation over Polygons”. en. In: *Proceedings of the 24th International Conference on Extending Database Technology (EDBT)*. OpenProceedings.org, 2021. DOI: 10.5441/002/EDBT.2021.16. URL: <https://openproceedings.org/2021/conf/edbt/p78.pdf>.
- [48] Jianliang Xu, Xueyan Tang, and Dik Lun Lee. “Performance analysis of location-dependent cache invalidation schemes for mobile environments”. In: *IEEE Transactions on Knowledge and Data Engineering* 15.2 (Mar. 2003), pp. 474–488. ISSN: 1041-4347. DOI: 10.1109/tkde.2003.1185846. URL: <http://dx.doi.org/10.1109/TKDE.2003.1185846>.